UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

SISTEMA DE RECOLECCIÓN, ETIQUETADO Y RECUPERACIÓN DE TEXTOS FINANCIEROS

Hugo Ferreira Moreno
Tutor: Pablo Alfonso Haya Coll
Ponente: Germán Montoro Manrique

Julio 2019

SISTEMA DE RECOLECCIÓN, ETIQUETADO Y RECUPERACIÓN DE TEXTOS FINANCIEROS

AUTOR: Hugo Ferreira Moreno
TUTOR: Pablo Alfonso Haya Coll

Dpto. Ingeniería informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2019

Resumen

El presente Trabajo Fin de Grado nace dentro del marco del proyecto Financial Text Analytics In Spanish: Tools and Language Resources siendo una colaboración conjunta de la Cátedra de Lingüística Computacional y el Departamento de Informática de la Universidad Autónoma de Madrid. Este proyecto persigue explorar las posibilidades del etiquetado semántico en textos financieros.

La finalidad de este Trabajo Fin de Grado es el etiquetado, extracción y recuperación de información de textos financieros sobre las memorias anuales de empresas con sede en España. Estos documentos públicos han sido recogidos de los sitios web de las empresas.

Una vez convertidas las memorias a texto plano, se aplican algoritmos de Procesamiento de Lenguaje Natural para la normalización y segmentación de estos documentos, con el objeto de extraer la información relevante diferenciada para su posterior clasificación y etiquetado. Se anotará semánticamente el texto resultante, y se almacenará indexado en una base de datos no relacional que, adicionalmente, será accesible online, de manera que se puedan consultar y recuperar los documentos.

Para poder llegar a cabo este proyecto, se ha organizado el Trabajo Fin de Grado en tres bloques. En primer lugar, se ha creado un modelo de procesamiento de textos para la extracción de keywords, basado en algoritmos de grafos y desarrollado en Python.

Con la ayuda de dos analizadores sintácticos (Grampal y Spacy) y siguiendo la línea de pensamiento del texto “TextRank: Bringing Order into Texts, Rada Mihalcea and Paul Tarau, Department of Computer Science, University of North Texas” usaremos los algoritmos de clasificación basados en grafos, que son capaces mediante la extracción de grupos sintácticos de determinar la importancia de sus nodos/vértices basándose en información global del texto recopilada recursivamente del grafo. Consiguiendo de esta manera resultados competitivos con el estado de arte desarrollado en las áreas relacionadas con las aplicaciones de lenguaje natural.

En segundo lugar, se usarán los índices de la herramienta BabelNet para extraer la categoría conceptual de cada keyword resultante del análisis anteriormente comentado.

Por último, guardaremos en una base de datos no relacional (elasticsearch) los datos extraídos y realizaremos el etiquetado semántico así como las categorías conceptuales de los textos analizados.

Palabras clave

Recopilación, etiquetado, extracción, grafo, procesamiento, lenguaje, texto, financiero

Abstract

The present Degree Final Project arises within the framework of the project Financial Text Analytics In Spanish: Tools and Language Resources being a joint collaboration of the Chair of Computational Linguistics and the Department of Computing of the Universidad Autónoma de Madrid. This project aims to explore the possibilities of semantic labeling in financial texts.

The purpose of this Degree Final Project is the labeling, extraction and retrieval of information from financial texts on the annual reports of companies based in Spain. These public documents have been collected from the websites of these companies.

Once the memories have been converted into plain text, Natural Language Processing algorithms are applied for the normalization and segmentation of these documents, in order to extract the differentiated relevant information for later classification and labeling. The resulting text will be semantically annotated, and it will be stored indexed in a non-relational database that, additionally, will be accessible online, so that documents can be consulted and retrieved.

In order to achieve this project, the Degree Final Project has been organized in three blocks. First, a word processing model has been created for the extraction of keywords, based on graph algorithms and developed in Python.

With the help of two syntactic analyzers (Grampal and Spacy) and following the line of thought of the text "TextRank: Bringing Order into Texts, Rada Mihalcea and Paul Tarau, Department of Computer Science, University of North Texas" we will use classification algorithms based in graphs, which are able by extracting syntactic groups to determine the importance of their nodes / vertices based on global information of the text recursively collected from the graph. In this way, competitive results are achieved with the state of the art developed in the areas related to natural language applications.

Secondly, the indexes of the BabelNet tool will be used to extract the conceptual category of each keyword resulting from the aforementioned analysis.

Finally, we will store the extracted data in a non-relational database (elasticsearch) and we will carry out the semantic labeling as well as the conceptual categories of the texts analyzed.

Keywords

Compilation, tagging, extraction, graph, processing, language, text, financial

Agradecimientos

Quiero agraceder a mi familia y amigos el apoyo brindado a lo largo de toda la carrera. Me gustaría agradecer también las facilidades proporcionadas por mi empresa y los ánimos de mis compañeros de trabajo.

Agradecer en especial a mi tutor, Pablo Haya, toda la ayuda prestada a lo largo del desarrollo del proyecto, sin tu paciencia esto no sería posible.

Gracias a todos.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

1.1 Motivación

En el mundo actual se genera alrededor de 2,5 quintillones de bytes a diario. Esto abarca información de toda clase, desde un simple comentario en las redes sociales a informes o libros enteros, o desde una pequeña imagen a todas las transmisiones a nivel global del IoT (Internet of Things).

Se estima que en 2020, cada persona generará alrededor de 1,7 MB de datos por segundo[1]. Si estamos en una población de unos 7.600 millones de personas en 2020[2] y alrededor de un 60%[3] de usuarios activos entre esa población esto supondría 7.752.000.000 MB por segundo.

De esta cantidad ingente de datos, el 80 % de ellos son no estructurados[4], es decir, que están expresados en lenguaje natural (hablado, escrito o visual) que un humano puede comprender fácilmente pero las aplicaciones tradicionales no.

Ante esta problemática, emerge una nueva generación de sistemas tecnologías y algoritmos, con el objetivo de almacenar, procesar, analizar y extraer la información relevante entre toda esta marea de datos. Estas tecnologías son el Data Mining y el Big Data. Esta última se presenta como una tecnología de apoyo, dado que la capacidad de almacenamiento de datos ha crecido enormemente, sin embargo el procesamiento y utilización de los mismos no han ido a la par[5].

Con todos estos datos, nosotros nos vamos a centrar en la parte referente a información textual (no estructurada), más específicamente textos financieros obtenidos de las memorias financieras publicadas por organizaciones privadas. Estos textos pueden contener información vital para una empresa o como en nuestro caso para una labor de investigación.

En concreto, hemos acotado el estudio a uno de los campos que está tomando cada vez más relevancia, el etiquetado semántico, que consiste en la categorización y etiquetado de estos textos permitiendo posteriormente analizarlos para sacar lo que sea relevante para nuestros propósitos.

Aunque ya existen numerosas herramientas de clasificación y etiquetado, estas a menudo son bastante pobres, ya que muchas de ellas dan soporte únicamente en inglés y al resto de idiomas le asocian una serie de traducciones, que tiene como consecuencia una peor calidad de los resultados, como ocurren en español.
Adicionalmente aparte de la calidad y cantidad de los datos, muchas de ellas son incapaces de extraer información sintáctica o de aplicar diferentes algoritmos de extracción de palabras clave (*keywords*).

[1] Data Never Sleeps 6.0*:* https://www.domo.com/learn/data-never-sleeps-6

[2] PopulationPyramid: https://www.populationpyramid.net/es/mundo/

[3] Digital 2019: https://wearesocial.com/blog/2019/01/digital-2019-global-internet-use-accelerates

[4] IBM Secutiry,2016 https://www.ibm.com/blogs/transformacion/2016/06/07/la-era-cognitiva-en-la-industria-de-la-salud/

[5] Data Mining : https://www.campusbigdata.com/big-data-blog/item/82-data-mining-vs-big-data

1.2 Objetivos

El objetivo de este proyecto consiste en crear un sistema de recolección, etiquetado y recuperación de palabras clave e información semántica en textos financieros. Para ello dividiremos la tarea en tres objetivos: obtención de las palabras claves, extracción de conceptos y etiquetado del texto.

En primer lugar, después de normalizar el texto, nos centraremos en los términos simples o compuestos que mejor representan la temática del texto. Se han probado diferentes métodos de evaluación, como frecuencia por palabras y varias configuraciones de TextRank (Mihalcea & Tarau, 2004). Al finalizar esta parte habremos extraído las palabras claves del texto.

En segundo lugar, nos centraremos en la extracción de conceptos de las palabras claves obtenidas. Esto se hará a través de Babelnet[6], que es al mismo tiempo una red semántica multilingüe y una ontología lexicalizada que integra Wikipedia (la enciclopedia multilingüe más amplia) y Wordnet[7] (una de las bases de datos con más información léxica en inglés). La información de la que dispone Babelnet se encuentra en un conjunto de archivos que ellos denominan índices, realizaremos la extracción de los índices que almacena la información semántica del español.

Una vez hemos extraído los índices, los almacenaremos, sin embargo para poder trabajar con un conjunto de datos tan grande necesitamos un motor que lo gestione. Para esta tarea el motor elegido es Elasticsearch[8], motor de búsquedas basado en Apache Lucene que realizará el indexado de la información y permitirá mediante consultas (querys) la extracción de la información deseada.
Elasticsearch solo admite un tipo de fichero, ficheros JSON por lo tanto se procederá a una transformación de los índices de babelnet a un formato legible por el sistema.
Una vez todos los documentos estén correctamente transformados e indexados obtendremos un sistema de almacenamiento con un índice por cada texto de todas sus palabras claves y conceptos derivadas de las mismas, pudiendo así realizar el objetivo principal de nuestro proyecto, obtener de forma automática la clasificación de cualquier texto que le pasemos y su indexado para posteriores búsquedas.

Adicionalmente, con el motor de búsqueda implantado, podremos hacer consultas a lo largo de todos los textos que integremos en el sistema, siendo capaces de buscar por ejemplo según un concepto específico (banca) o un conjunto de ellos (banca, inversión, economía).

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Introducción de la problemática planteada por este Trabajo Fin de Grado, y los objetivos marcados para su resolución.
- Estado del arte donde introducimos los conceptos claves así como el marco actual del proyecto.
- Diseño que hemos realizado del proyecto y los módulos que constituyen este proyecto.
- Desarrollo donde se explica el funcionamiento de los algoritmos más relevantes así como la integración de las diferentes módulos del proyecto.

[6] Babelnet: http://wwwusers.di.uniroma1.it/~navigli/pubs/AIJ_2012_Navigli_Ponzetto.pdf

[7] WordNet https://wordnet.princeton.edu/

[8] Elasticsearch: https://www.elastic.co/es/products/elasticsearch

- Pruebas que se han ido realizando para la validación, corrección y mejora del sistema desarrollado.
- Calidad del sistema creado mediante la aplicación de diferentes herramientas.

2 Estado del arte

2.1 Análisis sintáctico

El análisis sintáctico consiste en analizar la función sintáctica, las relaciones de concordancia así como la jerarquía que guardan las palabras en una oración. A menudo, la interpretación y comprensión de un texto dependen de un análisis sintáctico correcto.

Generalmente, dado la dificultad de definir el límite entre sintaxis y morfología se habla directamente de análisis morfosintáctico.

Los analizadores sintácticos esperan como entrada un componente léxico llamado comúnmente token por ello, previo a este análisis, entra en juego un conjunto de procesos que preparan el texto. Entre los cuales se encuentran la segmentación del texto en sentencias, la separación en tokens y la creación de los mismos a través de un analizador lexicográfico.

Por regla general, al menos en la lengua española, solo separan los tokens por espacios y signos de puntuación. Es posible a posteriori afinar los tokens obtenidos a través de un post-procesado (véase ejemplo en Figura 1).

Una vez los tokens generados, el análisis sintáctico podrá asignar una etiqueta morfosintáctica por cada token recibido.

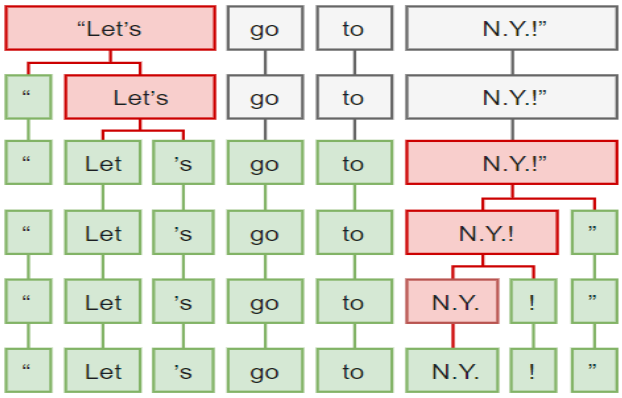


Figura 1: Ejemplo de análisis léxico con post-procesado

Como resultado de este análisis pasaremos de tener un texto formado de palabras a una lista de sentencias con sus tokens asociados, sean individuales o compuestos (con un post-procesamiento) que será la entrada para los algoritmos de clasificación elegidos.

Esta parte la realizan los dos analizadores sintácticos estudiados, Grampal y Spacy. Grampal es un analizador sintáctico del español que dispone de servicio web y Spacy es una biblioteca libre para el procesamiento del lenguaje natural en Python.
No se disponen los datos comparativos de Grampal, pero en Spacy obtenemos resultados que rivalizan con el estado de arte actual en segmentación de sentencias y extracción de tokens, tanto a nivel de precisión como a nivel de velocidad. En la tabla 1 y 2 se puede observar una comparativa del rendimiento de Spacy con otros algoritmos también en el estado del arte.

Sistema	Año	Tipo	Precisión
spaCy v2.0.0	2017	neural	94.48
spaCy v1.1.0	2016	lineal	92.80
Dozat and Manning	2017	neural	**95.75**
Andor et al.	2016	neural	94.44
SyntaxNet Parsey McParseface </>	2016	neural	94.15
Weiss et al.	2015	neural	93.91
Zhang and McDonald	2014	lineal	93.32
Martins et al.	2013	lineal	93.10

Figura 2: Comparativa de precisión de diferentes sistemas (parseo) para datos extraídos del Wall Street Journal, Penn Treebank[9]

	Valores absolutos			Valores relativos(a spacy)		
Sistema	Creación de:			Creación de:		
	Token	Etiqueta	Analisis	Token	Etiqueta	Analisis
spaCy	0.2ms	1ms	19ms	1x	1x	1x
CoreNLP	0.18ms	10ms	49ms	0.9x	10x	2.6x
ZPar	1ms	8ms	850ms	5x	8x	44.7x
NLTK	4ms	443ms	*n/a*	20x	443x	*n/a*

Figura 3: Comparativa de tiempos de diferentes sistemas[10]

Pese a la potencia del análisis sintáctico el conjunto de estas tareas por sí mismas no nos proporcionan el significado de las palabras. Para tratar la parte del significado de la palabra entre en juego el análisis semántico.

2.2 Análisis semántico

El análisis semántico es el proceso de relacionar estructuras sintácticas, como puede ser una sentencia o una palabra, con su significado. En el contexto de machine learning, nombre conocido para el conjunto de métodos de análisis de datos que automatiza la construcción de modelos analíticos, este se encarga de la creación de estructuras que aproximan conceptos al contenido. Para ponernos en perspectiva el cerebro humano es capaz de procesar el lenguaje de una forma compleja, siendo capaz de entender las palabras por, entre otras cosas, su experiencia, su forma, su contexto, su coherencia con el resto del texto. En definitiva un conjunto de técnicas que nos son innatas gracias a nuestras redes neuronales que son incluso capaces de detectar cosas tan sutiles

[9] Comparaciones de analizadores :https://www.aclweb.org/anthology/P15-1038

[10] Spacy docs : https://spacy.io/usage/facts-figures

para el procesamiento del lenguaje como pueden ser el sarcasmo, los dobles sentidos, juegos de palabras etc.

En las ciencias de la computación, la respuesta por lo tanto no es nada obvia y para su resolución se han probado varios métodos, entre ellos desde los más sencillos como la clasificación por frecuencias a aquellos basados en diccionarios con anotaciones manuales.

En una etapa más actual se derivan dos corrientes dentro las técnicas de machine learning: métodos completamente automáticos (no supervisados) a métodos supervisados con clasificadores que hoy en día son aquellos con más porcentaje de acierto hasta la fecha.

Todos estos conjuntos de técnicas se agrupan en lo que llamamos problemas WSD (*Word-sense disambiguation*) y WSI (*Word-sense induction*).

2.2.1 Word Sense Disambiguation y Word Sense Induction

Word Sense Disambiguation (WSD) (Desambiguación del sentido de la palabra) es la tarea encargada de la clasificación de tokens en un determinado contexto en una o varias categorías.

Para ilustrarlo, veamos uno de los ejemplos más populares, con el token “banco” en las siguientes frases “Me siento en el banco” y “Voy al banco a sacar dinero”. La WSD permitiría diferenciar dos resultados, banco (objeto, mueble), y banco (Institución Financiera).

Una variante más reciente es *Word Sense Induction (WSI)* (Inducción del sentido de la palabra) en el cual las clases de sentidos no están predefinidos sino que extraerá el sentido desde los datos que le proporcionamos usando técnicas de agrupamiento no supervisadas, los agrupamientos formados definirán el sentido de la palabra en ese contexto particular.

Hay muchos estudios que consideran las tareas de WSD/WSI uno de los mayores desafíos para la Inteligencia Artificial, ya que no solo implica conocimiento lingüístico general sino que también requiere conocimiento específicos de las palabras. Por ejemplo retomando el ejemplo anterior “Me siento en el banco” deducimos que el banco se refiere al mueble pero podría ser que nos refiramos a banco como edificio.

La combinación de estas técnicas WSD y WSI requieren dos entradas: un diccionario en el que se específica las palabras de las que se quiere extraer el sentido específico en el contexto, y un corpus de lenguaje de datos. Adicionalmente según el método utilizado puede incluir un corpus de entrenamiento también.

Desgraciadamente nuestros conocimientos no están todavía en un formato interpretable por los computadores por lo que se sigue trabajando actualmente en la conversión a un formato legible (ej. tripletas de relación sujeto-relación-objeto).

A pesar de la dificultad en la interpretación varios proyectos condensan esta clase de información creando lo que se conoce como redes semánticas y ontologías lexicalizadas.

2.2.2 Babelnet

Existen varias ontologías en español en la web pero hoy por hoy son mucho más escasas que en inglés, lo cual dificulta enormemente el encontrar una ontología de calidad.

Proyectos como BabelNet[11] están centrados en integrar las mayores bases de conocimientos de las que disponemos ahora mismo a nivel mundial (Wikipedia, Wordnet) junto con otra redes de conocimiento (EuroWordNet, MultiWordNet ...)
Creando debido a la cantidad de datos enlazados y a la calidad de los mismos una enorme red semántica multilingüe que sirve como referente a nivel mundial para más de 284 idiomas.
BabelNet ha sido capaz de establecer relaciones entre lo que ellos definen como *synset* (concepto), sus *sense* (sentidos) entre ellos y clasificarlos si cabe en alguna de sus categorías.

BabelNet no solo ha recogido la información en varios idiomas de diferentes ontologías sino que siendo consciente de las diferencias en la cantidad y calidad de la información ha implementado una solución.
Esta consiste en una serie de mecanismos de automatización para la traducción de la información, esto implica que la información de un idioma pueda ser ampliada por conocimientos que están en un idioma diferente.

Por todo ello este sistema implementado les otorgó el premio al *prominent paper award* en 2017 y cobertura mediática en medios tan conocidos como *The guardian.*

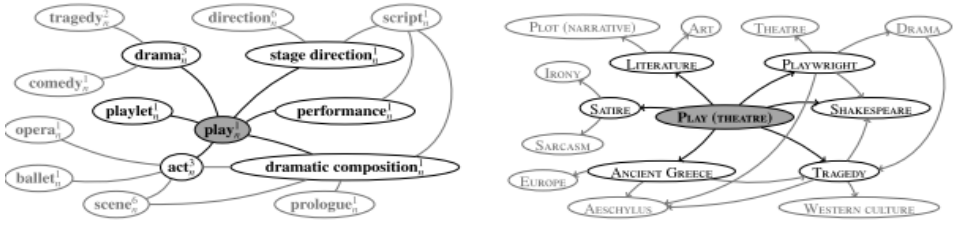
A continuación exponemos la arquitectura así como una breve descripción de sus principales atributos (*synset* y *sense),* atributos que se utilizarán durante el desarrollo del proyecto.

$\{\text{play}_n^8, \text{child's play}_n^2\}$.

Figura 4: Ejemplo de synset de BabelNet

Así es como se vería por ejemplo uno de las posibles relaciones del *synset* "jugar", estaría relacionado con juego de niños.

Juntando los *synsets* y sus relaciones se puede establecer una correspondencia de sentidos asociadas a las palabras, a modo de grafo representativo obtendremos lo siguiente:



(a) Excerpt of the WordNet graph centered on the synset play_n^1.

(b) Excerpt of the Wikipedia graph centered on the Wikipage Play (theatre).

Figura 5: Ejemplo de grafo asociativo en Babelnet (Wiki y Wordnet)

En definitiva, han sacado de Wordnet todos los conceptos (*word senses*) y sus relaciones léxicas y semánticas entre estos *synsets* con los datos de Wikipedia. Las páginas de Wikipedia (wikipages) formarían los conceptos y los links con otras páginas (wikilinks) formarían sus relaciones.

Un ejemplo de este mapeado de la red semántica sería el siguiente:

[11] Babelnet: http://wwwusers.di.uniroma1.it/~navigli/pubs/AIJ_2012_Navigli_Ponzetto.pdf

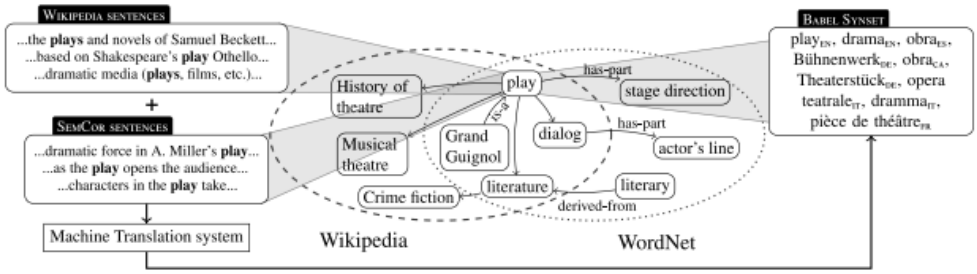


Figura 6: Grafo ilustrativo del etiquetado/relaciones de Babelnet

En BabelNet se extrajeron los rendimientos de diferentes algoritmos de WSD. Antes de mostrar los resultados, vamos a explicar los términos que se usan para los comparativas entre los diferentes métodos.

Precisión: Se define como la fracción de documentos recuperados que son relevantes para la consulta.
Y su fórmula general es:

$$\text{Precisión} = \frac{|\{\text{documentos relevantes}\} \cap \{\text{documentos recuperados}\}|}{|\{\text{documentos recuperados}\}|}$$

Cobertura (Recall): Se define como la fracción de documentos relevantes que han sido recuperados satisfactoriamente.
Y su fórmula general es:

$$\text{Exhaustividad} = \frac{|\{\text{documentos relevantes}\} \cap \{\text{documentos recuperados}\}|}{|\{\text{documentos relevantes}\}|}$$

F-measure: Se define como la media armónica de la precisión y de la exhaustividad.
Y su fórmula general es:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

	Mapping method		P	R	F_1	A
BoW	taxonomic		**89.7**	47.8	62.3	72.6
	gloss		87.6	51.8	65.1	74.0
	taxonomic + gloss		87.5	65.6	75.0	80.9
Graph	taxonomic relations					
	max depth	@ 2	87.2	60.8	71.6	77.9
		@ 3	81.6	65.0	72.4	78.7
		@ 4	78.3	69.5	73.6	79.4
	gloss relations					
	max depth	@ 2	80.5	60.6	69.1	77.0
		@ 3	77.5	65.2	70.9	78.2
		@ 4	72.4	67.1	69.6	78.0
	taxonomic + gloss relations					
	max depth	@ 2	81.2	74.6	**77.7**	**82.7**
		@ 3	72.8	**77.4**	75.1	80.1
		@ 4	64.3	76.2	69.8	75.0
	MFS baseline		25.4	49.2	33.5	25.4
	Random baseline		24.2	46.9	31.9	24.2

Figura 7: Comparativa de rendimiento de los diferentes métodos de mapeo

	Mapping method		Accuracy	
			w/o MFS	+ MFS
BoW	taxonomic		59.4	73.4
	gloss		43.5	70.0
	taxonomic + gloss		64.3	76.8
Graph	taxonomic relations			
	max depth	@ 2	66.2	72.5
		@ 3	69.1	71.5
		@ 4	71.0	72.5
	gloss relations			
	max depth	@ 2	59.9	69.6
		@ 3	60.9	66.2
		@ 4	60.9	63.8
	taxonomic + gloss relations			
	max depth	@ 2	**78.7**	**80.7**
		@ 3	74.9	75.4
		@ 4	68.6	69.1
	MFS baseline		64.7	
	Ponzetto & Navigli [105]		73.9	
Toral et al. [126]	best unsupervised		64.7	
	best supervised*		77.7	
	best voting		68.0	
	best unsupervised combination		65.7	
	best supervised combination*		77.2	
	oracle (upper bound)		84.5	

Figura 8: Comparativa de rendimiento de precisión de mapeos diferenciados por algoritmos

Los resultados muestran que el sistema implementado por Ponzetto & Navigli (*taxonomic + gloss relations*) alcance casi un 78% de *F1*, con rendimientos competitivos con el estado de arte actual (en obtención de *synsets* Tabla 3 y conceptualización Tabla 4).

Comparándola directamente contra otras bases de conocimiento también obtienen buenos resultados. Sobre todo teniendo en cuenta que KnowNet-20[12] y TSSEM (Cuadros, Rigau, y Castillo, 2007), son a la fecha los recursos con mejor rendimiento (sin tener datos de anotaciones manuales de los sentidos) en diferentes conjuntos de evaluación (Senseval y SemEval)

(a) Senseval-3 English Lexical Sample task:

Knowledge base	P	R	F_1	Avg. size
TRAIN	65.1	65.1	65.1	450
TRAIN-MFS	54.5	54.5	54.5	–
WN-MFS	53.0	53.0	53.0	–
SEMCOR-MFS	49.0	49.1	49.0	–
TSSEM	52.5	52.4	52.4	103
BabelNet-1	44.3	44.3	44.3	119
BabelNet-2	35.0	35.0	35.0	2,128
KnowNet-20	44.1	44.1	44.1	610
RANDOM	19.1	19.1	19.1	–

(b) SemEval-2007 English Lexical Sample (task 17):

Knowledge base	P	R	F_1	Avg. size
TRAIN	87.6	87.6	87.6	450
TRAIN-MFS	81.2	81.2	81.2	–
WN-MFS	66.2	59.9	62.9	–
SEMCOR-MFS	42.4	38.4	40.3	–
WN + XWN + KN-20	53.0	53.0	53.0	627
BabelNet-1	52.2	46.3	49.1	130
BabelNet-2	56.9	53.1	54.9	2,352
KnowNet-20	49.5	46.1	47.7	561
RANDOM	19.1	19.1	19.1	–

Figura 9: Evaluación de rendimiento de bases de conocimientos

[12] KnowNet: http://adimen.si.ehu.es/web/knownet

2.3 Extracción de palabras claves

Para entrar más en detalle con la primera parte del proyecto, describiremos las principales líneas de pensamiento al respecto y el estado del arte actual así como una comparativa con lo desarrollado.

Definimos la extracción de palabras claves (*keyword extraction*) como la identificación de un término o términos que mejor describen un texto, representando así la información más relevante del texto. Según el objetivo del análisis de los datos, los algoritmos utilizados se clasifican en supervisados y no supervisados (Weiss y Indurkhya, 1998).

En términos generales los algoritmos supervisados se les ha dado un enfoque basado en los problemas de clasificación binaria mientras que los no supervisados se dividen en tres sub-corrientes: *TF-IDF, clustering y graph-based ranking*.

Los algoritmos supervisados requieren grandes cantidades de datos previamente etiquetados (training data) y son dependientes del dominio específico en el que se ha hecho el entrenamientos de los textos, fuera de este contexto los resultados son muy pobres.

Las características más importantes para estos sistemas son la frecuencia y la localización del término en el conjunto, usando métodos basados en aprendizaje automático. Estos componen la base de lo que se considera hoy el estado del arte en algoritmos supervisados.

A estos se les incorporó conocimientos lingüísticos (Hulth, 2003), asociaciones estadísticas (Turney 2003) y información semántica (KEA++ 2006). En una corriente más actual aparecen métodos de *Support Vector Machines (SVM)* que junto con nuevas técnicas de *Natural Language Processing (NLP)* marcan la referencia actual en métodos supervisados.

Por otro lado, los algoritmos no supervisados tienen menor precisión en dominios específicos pero no son dependientes de un solo dominio, además no requieren de un entrenamiento previo por ello ante la creciente cantidad de datos disponibles esta corriente está ganando más peso.

En este estudio nos centraremos en estos últimos, describiremos el estado del arte y usaremos los algoritmos supervisados simplemente como punto de comparación.

2.3.1 Algoritmos no supervisados

Los algoritmos no supervisados han tenido varias aproximaciones desde su creación. Una de las grandes corrientes y la implementada en este Trabajo Fin de Grado son los algoritmos basados en grafos. Para ser rigurosos con el estado de arte actual describiremos las otras dos aproximaciones usadas en la actualidad, las técnicas basadas en agrupaciones (*clustering*) y las técnicas de incrustación (*embedding*).

2.3.1.1 Clustering

Estos algoritmos se basan en la agrupación (*clustering*) de sentencias para destacar las partes del texto que están semánticamente relacionadas, estas frases reflejarán mejor el tema del documento, de las partes destacadas se agrupan las palabras clave que reflejarán se extraerán las *keyphrase* y *keywords* respectivamente.

Esta labor se realiza mediante la similitud de vectores sentencia (las sentencias y las palabras se transforman en vectores), que usan un diverso conjunto de técnicas.

Entre las más utilizadas, técnicas de co-ocurrencia, que agrupan por ocurrencia de un término con respecto a otro según su distancia, y técnicas más complejas como puede ser el *Markov Cluster Process* (MCP)[13], basado en matrices de probabilidad de proximidad.

2.3.1.2 Embedding

Desde que en 2013 nació la idea de incrustación de palabras (Mikolov et al 2013), se produjo un gran avance dado la introducción de una representación de las palabras.
Las palabras estarían transformadas como vectores en un espacio vectorial posibilitando técnicas matemáticas avanzadas.

Esta idea permitió ir un paso más allá creando algoritmos que ahora constituyen el estado de arte actual en extracción de palabras clave (*WordAttractionRank*[14] y *EmbedRank*[15]).

Sin embargo, aunque son notables los avances con los algoritmos de embedding, hemos decidido basar el proyecto en uno de los algoritmos que a día de hoy sigue siendo el pilar de los algoritmos basados en grafos, *Text rank* (Rada Mihalcea and Paul Tarau, 2004).

N	Method	Inspec			DUC			NUS		
		P	R	F_1	P	R	F_1	P	R	F_1
5	TextRank	24.87	10.46	14.72	19.83	12.28	15.17	5.00	2.36	3.21
	SingleRank	38.18	23.26	28.91	30.31	19.50	23.73	4.06	1.90	2.58
	TopicRank	33.25	19.94	24.93	27.80	18.28	22.05	16.94	8.99	11.75
	Multipartite	34.61	20.54	25.78	29.49	19.42	23.41	**19.23**	**10.18**	**13.31**
	WordAttractionRank	38.55	23.55	29.24	30.83	19.79	24.11	4.09	1.96	2.65
	EmbedRank d2v	**41.49**	**25.40**	**31.51**	30.87	19.66	24.02	3.88	1.68	2.35
	EmbedRank s2v	39.63	23.98	29.88	**34.84**	**22.26**	**27.16**	5.53	2.44	3.39
	EmbedRank++ s2v ($\lambda = 0.5$)	37.44	22.28	27.94	24.75	16.20	19.58	2.78	1.24	1.72
	EmbedRank$_{positional}$ s2v	38.84	23.77	29.49	39.53	25.23	30.80	15.07	7.80	10.28
10	TextRank	22.99	11.44	15.28	13.93	16.83	15.24	6.54	6.59	6.56
	SingleRank	34.29	39.04	36.51	24.74	30.97	27.51	5.22	5.04	5.13
	TopicRank	27.43	30.8	29.02	21.49	27.26	24.04	13.68	13.94	13.81
	Multipartite	28.07	32.24	30.01	22.50	28.85	25.28	**16.51**	**17.36**	**16.92**
	WordAttractionRank	34.10	38.94	36.36	25.06	31.41	27.88	5.15	5.12	5.14
	EmbedRank d2v	**35.75**	**40.40**	**37.94**	25.38	31.53	28.12	3.95	3.28	3.58
	EmbedRank s2v	34.97	39.49	37.09	**28.82**	**35.58**	**31.85**	5.69	5.18	5.42
	EmbedRank++ s2v ($\lambda = 0.5$)	30.31	34.29	32.18	18.27	23.34	20.50	1.91	1.69	1.79
	EmbedRank$_{positional}$ s2v	32.46	36.61	34.41	32.23	39.95	35.68	13.50	13.36	13.43
15	TextRank	22.80	11.50	15.29	11.25	19.21	14.19	6.14	9.16	7.35
	SingleRank	30.91	48.92	37.88	21.20	38.77	27.41	5.42	8.24	6.54
	TopicRank	24.51	37.45	29.62	17.78	32.92	23.09	11.04	16.47	13.22
	Multipartite	25.38	41.32	31.44	19.72	36.87	25.7	**14.13**	**21.86**	**17.16**
	WordAttractionRank	30.74	48.62	37.66	21.82	40.05	28.25	5.11	7.41	6.05
	EmbedRank d2v	31.06	48.80	37.96	22.37	40.48	28.82	4.33	5.89	4.99
	EmbedRank s2v	**31.48**	**49.23**	**38.40**	**24.49**	**44.20**	**31.52**	5.34	7.06	6.08
	EmbedRank++ s2v ($\lambda = 0.5$)	27.24	43.25	33.43	14.86	27.64	19.33	1.59	2.06	1.80
	EmbedRank$_{positional}$ s2v	29.44	46.25	35.98	27.38	49.73	35.31	12.27	17.63	14.47

Figura 10: Evaluación de diferentes métodos de extracción de keywords

2.3.1.3 Graph-based

Los algoritmos basados en grafos, transforman el texto en un grafo y se fundamentan en un sistema de recomendaciones. Siguiendo las conexiones (aristas) se le da más o menos importancia a un

[13] MCP https://sites.cs.ucsb.edu/~xyan/classes/CS595D-2009winter/MCL_Presentation2.pdf

[14]WordAttractionRankhttps://pdfs.semanticscholar.org/bd37/94c777af5ba363abae5708050ea78ecc97e2.pdf

[15] EmbedRank: https://arxiv.org/abs/1801.04470

nodo en función del número de conexiones y de la importancia de los nodos que se conectan con él. El total de esta información global es extraída recursivamente de todo el grafo.

Como estandarte de estos algoritmos tenemos a *Text rank*, un algoritmo fácil de implantar, que es capaz de seguir compitiendo con el estado del arte en la extracción de *keywords*.
Observando la tabla 6 se puede observar que sobre los datos del conjunto NUS tiene resultados superiores a *EmbedRank*.

Una vez generado el grafo, el algoritmo recuperará la importancia del nodo en función de lo anteriormente expuesto y la aplicará una puntuación al nodo (o vértice) en base al conjunto del grafo, una vez iterado por todos los nodos tendremos un ranking de clasificación de los nodos.

Definimos la función de la puntuación de un vértice tal y como dieron Brin y Page ("The anatomy of a large-scale hypertextual Web search engine", 1998)

$$S(V_i) = (1-d) + d * \sum_{j \in In(V_i)} \frac{1}{|Out(V_j)|} S(V_j)$$

Entrando más en detalle en el algoritmo implantado *Text rank*, esta nueva fórmula tiene en cuenta el peso de las aristas al calcular la puntuación asociada a los vértices de un grafo, esta definición viene dada por la siguiente fórmula:

$$WS(V_i) = (1-d) + d * \sum_{V_j \in In(V_i)} \frac{w_{ji}}{\sum_{V_k \in Out(V_j)} w_{jk}} WS(V_j)$$

Para poder ejecutar el algoritmo, hace falta seguir los pasos descritos a continuación.
Se convierte el texto en tokens, se aplica un filtrado sintáctico de tokens, se lematizan sacando el lema constituyente del token, se identifican las relaciones determinando el coeficiente de correlación, es decir la distancia entre palabras.
Con estos pasos tendremos el grafo construida y solo queda aplicar el algoritmo *PageRank* para obtener la puntuación relativa a los nodos, finalmente se ordenan los resultados de mayor a menor, teniendo en los primeros lugares las palabras más relevantes del texto.

A continuación se presentan las pruebas sobre las configuraciones para *TextRank*.

Method	Assigned		Correct		Precision	Recall	F-measure
	Total	Mean	Total	Mean			
TextRank							
Undirected, Co-occ.window=2	6,784	13.7	2,116	4.2	**31.2**	43.1	**36.2**
Undirected, Co-occ.window=3	6,715	13.4	1,897	3.8	28.2	38.6	32.6
Undirected, Co-occ.window=5	6,558	13.1	1,851	3.7	28.2	37.7	32.2
Undirected, Co-occ.window=10	6,570	13.1	1,846	3.7	28.1	37.6	32.2
Directed, forward, Co-occ.window=2	6,662	13.3	2,081	4.1	31.2	42.3	35.9
Directed, backward, Co-occ.window=2	6,636	13.3	2,082	4.1	31.2	42.3	35.9
Hulth (2003)							
Ngram with tag	7,815	15.6	1,973	3.9	25.2	**51.7**	33.9
NP-chunks with tag	4,788	9.6	1,421	2.8	29.7	37.2	33.0
Pattern with tag	7,012	14.0	1,523	3.1	21.7	39.9	28.1

Figura 11: TextRank

2.4 Bases de datos no relacionales

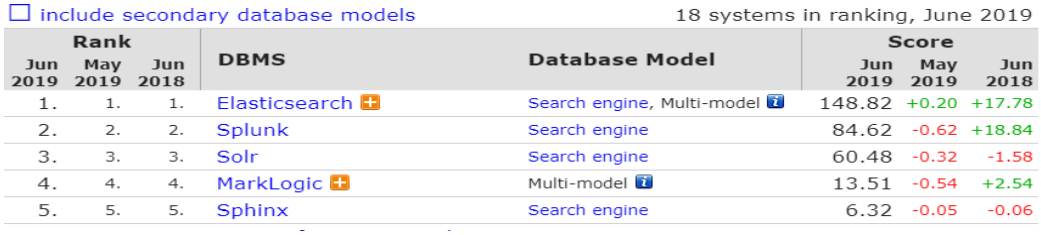
Como hemos comentado anteriormente la creciente cantidad de datos plantearon una serie de desafíos que los sistemas de tratamiento de datos que seguían el modelo clásico de
Sistema de Gestión de Bases de Datos Relacionales (SGBDR) no podían resolver.
Las exigencias de tiempo y rendimiento pasaban a un primer plano y las bases de datos tradicionales excedían el tiempo de procesado esperado para una respuesta en tiempo real.

A cambio de una pérdida de coherencia, un modelo de datos no relacional (NoSQL) ofrece una ganancia en escalabilidad y rendimiento. Estas bases de datos pueden tener diferentes tipos de representación clave-valor, documentos, gráficos…

Uno de los proyectos que trabaja con bases de datos NoSQL y que hoy en día es uno de los referentes para la implementación de motores de búsquedas dado su utilidad para la recuperación de información (indexado, búsquedas etc) es Lucene (Apache Lucene 1999).

Tal es su calado que dos de los mejores motores de búsqueda open source se basan en él:
Solr (2004) y Elasticsearch (2010). Este último servidor de búsqueda es el usado para albergar los resultados obtenidos de la extracción y conceptualización de los textos

Programado en Java, este motor de búsqueda de texto completo, que nació en 2010 ocupa actualmente el primer puesto en motores de búsqueda según db-engines[16], página web de referencia a nivel mundial en materia de bases de datos.



include secondary database models — 18 systems in ranking, June 2019

Rank			DBMS	Database Model	Score		
Jun 2019	May 2019	Jun 2018			Jun 2019	May 2019	Jun 2018
1.	1.	1.	Elasticsearch	Search engine, Multi-model	148.82	+0.20	+17.78
2.	2.	2.	Splunk	Search engine	84.62	-0.62	+18.84
3.	3.	3.	Solr	Search engine	60.48	-0.32	-1.58
4.	4.	4.	MarkLogic	Multi-model	13.51	-0.54	+2.54
5.	5.	5.	Sphinx	Search engine	6.32	-0.05	-0.06

Figura 12: Comparativas sobre motores de búsqueda

[16] Ranking de motores de búsqueda https://db-engines.com/en/ranking/search+engine

3 Diseño

El diseño del sistema se estructura en tres grandes bloques: A) obtención de palabras claves; B) extracción de conceptos, y C) etiquetado de texto. Aunque los diferentes módulos en su conjunto responden a la problemática planteada también pueden ser usados como módulos independientes.

Teniendo en cuenta estos bloques se diseña la arquitectura del sistema de la siguiente forma:

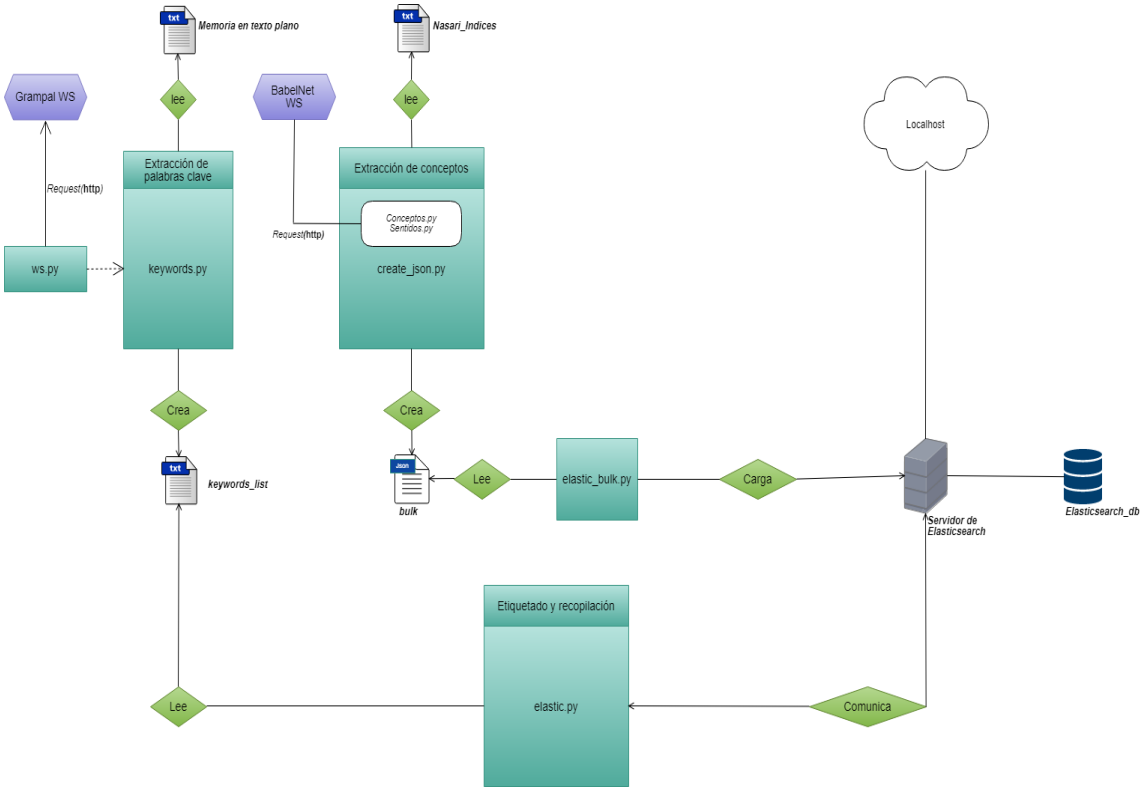


Figura 1: Diagrama de arquitectura del proyecto

3.1 Análisis

3.1.1 Extracción de palabras clave

En primer lugar, para la extracción de las palabras hemos analizado el funcionamiento de *TextRank*. A continuación se muestra la explicación de la idea general del algoritmo expuesta por sus propios autores (Rada Mihalcea and Paul Tarau, TextRank: Bringing Order into Texts 2004)[17].

Adaptándola a una situación práctica y desglosando el proceso en tareas concretas, se obtiene el siguiente listado de tareas:

1. Separación del texto en sentencias
2. Tokenización de las sentencias
3. Etiquetado de las partes de la sentencia (*Part-of-speech tagging*), de aquí lo que nos interesa es el lema y su categoría gramatical.

[17] TextRank: Bringing Order into Texts https://www.aclweb.org/anthology/W04-3252

4. Creación de un grafo, con las palabras pertenecientes a la categoría gramatical que queramos, los nodos serán las palabras.
5. Creación de los enlaces entre las palabras, traducido ahora al ámbito del grafo, creación de las aristas de los vértices, se creará una arista (no dirigida) cuando los nodos aparezcan separados a una distancia inferior o igual a un coeficiente *k*, que definiremos como coeficiente de co-ocurrencia.
6. Aplicaremos *PageRank*.
7. Ordenaremos los resultados.
8. Seleccionaremos los *T* nodos con mayores valores.
9. Se comprobará la existencia de términos conjuntos entre los *T* nodos mayores, y de haberlos se unirán en una solo entrada.
10. Se devolverá el resultado con las palabras claves obtenidas.

Finalmente para las pruebas de validación se probará el algoritmo con textos de memorias anuales recogidas manualmente.

Para estas tareas ha sido necesario estudiar la estructura de la respuesta de los analizadores morfosintácticos encontrados (Grampal y Spacy) y ver cómo se adaptan a nuestros objetivos.

De estos necesitamos tres informaciones: la tokenización, la lematización y la categoría gramatical.

3.1.1.1 Formatos

Por regla general se siguen una serie de formatos que son usados como base para todos los desarrollos en NLP, estas directrices tienen que estar presente en el diseño de los algoritmos a la hora de trabajar con informaciones contenidas en los mismos.

En este estudio no ha hecho falta profundizar con el formato seguido por Spacy al tener su propia API(Application Programming Interface) con un método "get" para los campos requeridos, aunque sí hemos tenido que estudiar los métodos involucrados[18].

Sin embargo para Grampal hemos tenido que estudiar el formato de respuesta.
Al funcionar como un servicio web se ha podido desarrollar un módulo propio (que ha servido para hacer pruebas de integración con el resto del proyecto).
Adicionalmente se ha testeado manualmente accediendo a la dirección donde está disponible el servicio (***http://leptis.lllf.uam.es/api/grampal/general)***

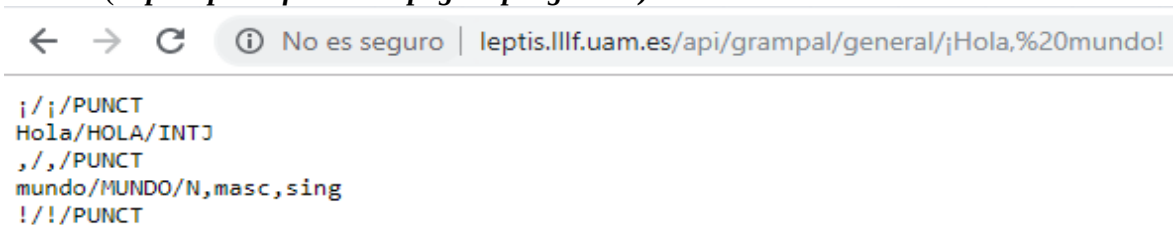


Figura 2: Ejemplo de análisis léxico con post-procesado

[18] Spacy doc:https://spacy.io/api/doc

La respuesta sigue un formato concreto, este es una adaptación de lo que se conoce como "Formato CoNLL"[19], desarrollado para la conferencia computacional del procesamiento del lenguaje natural (Conference on Computational Natural Language Learning). Con el objetivo de tener un formato de dependencias común para los diferentes corpus de datos.

Este formato impone los siguientes campos:
-*ID*: Índice de la palabra
-*Form*: Forma de la palabra (origen)
-*Lemma*: Lema
-*Upostag* (Universal part-of-speech tag): Etiqueta Gramatical
-*Xpostag* (Language-specific part-of-speech tag): extensión de la etiqueta Gramatical
-*Feats*: Lista de características morfológicas
-*Head*: Cabeza del token(un valor numérico asociado a un token o 0 si es raíz)
-*Deprel*: Etiqueta gramatical del "universal stanford dependency relation[20]"
-*Deps*: Lista de dependencias secundarias
-*Misc*: Otras anotaciones

La API de Grampal usa una versión reducida de este formato, en su respuesta se muestra la forma (origen)/lema/etiqueta de categoría gramatical y sus rasgos morfológicos, de estos últimos solo necesitaremos la categoría gramatical.

Existen otros formatos de representación como el usado por Stanford que persigue el mismo objetivo de estandarizar las representaciones para facilitar la integración de los diferentes corpus/treebanks.

CATEGORÍAS SINTÁCTICAS:				
Sustantivo	**N**	nombre	*gato*	
	NPR	nombre propio	*María*	
Adjetivo	**ADJ**		*azul*	
Verbo	**V**		*cantar*	
Auxiliar	**AUX**		*habría*	
Pronombre	**P**		*yo, ahora*	
	REL	relativo	*que*	
	PINT	interrogativo	*qué, cómo*	
Determinante	**ART**	artículo	*el*	
	POSS	posesivo	*su*	
	DEM	demostrativo	*ese*	
Cuantificador	**Q**		*uno*	cardinal
			primer	ordinal
			muchos	indefinido
Preposición	**PREP**		*en*	
Adverbio	**ADV**		*así*	
Conjunción	**C**		*y*	
Interjección	**INTJ**		*ole*	
Marcador discursivo	**MD**		*oye*	
Desconocida	**UNKN**		*xx*	

Figura 3: Categorías gramatical Grampal

[19] CoNLL-U format https://universaldependencies.org/format.html

[20] Stanford dependency relation https://universaldependencies.org/docs/u/dep/index.html

3.1.1.2 Grafos

Para la parte de generación del grafo hemos decido trabajar con `python-igraph` al tener incorporado las funcionalidades básicas para trabajar con grafos. Además esta librería incorpora algunos algoritmos que se pueden usar en los grafos que construyamos, entre ellos está *PageRank*.

3.1.1.3 Estructura

Motivado por las características del servicio de Grampal y para tener un mayor control se dividirá en dos módulos.

Un módulo principal donde se hará todo el proceso de lectura del texto, división por sentencias, normalización, tokenización, análisis sintáctico, creación de grafo. Adicionalmente aplicará la función de evaluación y creará un documento con los resultados. Por otro lado, se creará un módulo secundario que trabajará únicamente con Grampal.

La salida del módulo principal contendrá la siguiente información: Lema, valor del nodo al aplicar *TextRank*, apariciones, frecuencia de aparición, listado de formas en las que aparece (Ej:Negocios, negocio) y su categoría gramatical.

3.1.2 Extracción de conceptos

Con la idea del etiquetado automática en mente ya hemos visto cómo en una primera parte necesitaríamos extraer la información más representativa de un texto. Esto se ha hecho mediante una aproximación de extracción de palabras clave, palabras que tendrán un significado representativo por sí mismas del significado del texto.

Una vez que tenemos estas palabras clave seguimos sin tener forma de darle un etiquetado semántico. Para ello necesitamos un repositorio semántico con multitud de conceptos para poder contextualizar estas palabras clave y extraer sus conceptos.

El problema principal que expusimos anteriormente es la falta de calidad/cantidad de la mayoría de repositorios semánticos. Por ello hemos analizado los repositorios con más peso a nivel global para usarlos como base. Para realizar la extracción de conceptos, hemos estudiado, por ser los más ampliamente conocidos y los más completos EuroWordNet[21], Concepnet [22]y Babelnet.

3.1.2.1 *Estudio de viabilidad*

EuroWordNet solo provee los datos bajo licencias específicas, además el proyecto acabó en 1999 por lo que por la calidad de sus datos ofrecidos y por su falta de integración (no tiene API) la descartamos.

El siguiente estudio de viabilidad se hizo con Concepnet, una red semántica libre y con su propia API (en varios lenguajes de programación). Permite obtener las relaciones de una palabra, sus sinónimos, contexto… el problema reside en que en español tiene muy pocas referencias.
Se hicieron pruebas sobre términos concretos a través de su API online (http://api.conceptnet.io), para ilustrarlo podemos observar los resultados que se obtuvieron sobre el término banco/bank.

[21] EuroWordNet http://projects.illc.uva.nl/EuroWordNet/

[22] ConceptNet http://conceptnet.io/

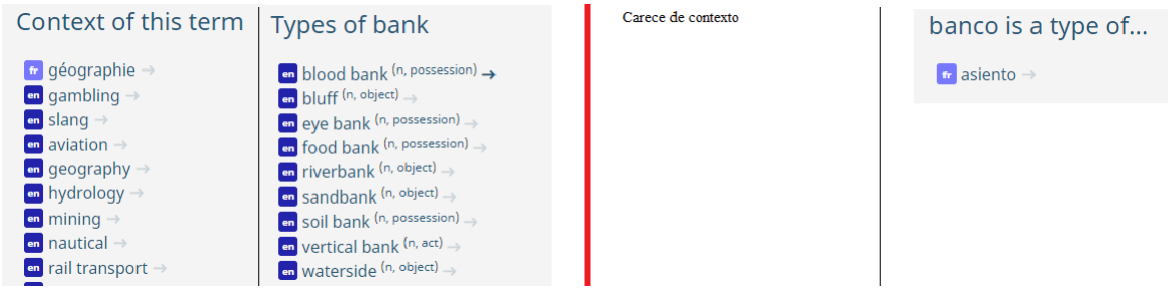


Figura 4: Comparativa Inglés/español Concepnet

A pesar de la riqueza de la respuesta para los términos en inglés (la figura 8 solo representa un pequeño extracto de la respuesta), la parte de datos en español apenas está alimentada.

3.1.2.2 *Babelnet*

Finalmente nos decantamos por Babelnet, que es al mismo tiempo la enciclopedia multilingüe más amplia hasta la fecha y una red semántica con más de 15 millones de entradas.

Se realizaron pruebas sobre su API online (http://live.babelnet.org/) retomando el ejemplo anterior tendríamos las siguientes categorías conceptuales para la palabra “banco”

Figura 5: Ejemplo de conceptualización en Babelnet

Adicionalmente, BabelNet incorpora su propia API en varios lenguajes de programación aunque tiene un límite de mil peticiones al día.

Pese a esta restricción en su API, Babelnet es de largo la mejor opción, además se enlaza con el proyecto NASARI[23] (Novel approach to semantically-aware representation of items).

3.1.2.3 *NASARI*

NASARI es un proyecto libre, en el cual se pueden descargar los índices que contienen las relaciones semánticas de los términos en español que usa BabelNet.

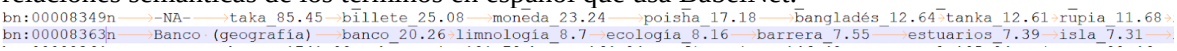


Figura 6: Fichero de Relaciones semánticas (Babelnet/NASARI)

[23] NASARI: http://lcl.uniroma1.it/nasari/

Con estos índices podemos buscar los diferentes conceptos de una palabra y su relación con el resto de las palabras que la definen. Los campos del fichero son los siguientes:

- *id*: id que tiene el término en babelnet (synset_id)
- *lema*: lema correspondiente en wikipedia
- *lista*: lista ordenada de clave_valor de conceptos constituyentes del lema.

Para dar un ejemplo más próximo, y entender un poco mejor los algoritmos de implementación de la parte del etiquetado hay que entender la forma en la que se ha representa los datos en estos índices.

El enfoque que le han dado es el siguiente: una palabra se define en función de una serie de palabras y una puntuación asociada a los mismos. La palabra banco por ejemplo se define como: banco (estructura) 100, banco (entidad bancaria) 100, banco (financiero) 100, banco (mueble) 90, banca 88, banquero 70, dinero 70, transacción 65, crisis 50 etc.

3.1.2.4 *Estructura de la extracción de conceptos*

En definitiva, tenemos dos formas de obtener nuestro repositorio.
La primera sería mediante consultas a BabelNet, que es un sistema muy potente, actualizado de forma continuada pero muy limitado debido al límite diario de peticiones.
La segunda mediante la implantación de nuestro propio repositorio basado en los índices de NASARI. Perderíamos en contenido, y en actualizaciones pero no tendríamos límites con respecto al número de peticiones además de tener un sistema ampliable y personalizable si se quisiera.

Valorando las ventajas y desventajas de las dos soluciones se decidió dividir esta tarea en dos módulos. Uno que trabaje directamente con la API de BabelNet, que pueda servir para añadir información de *synsets* de babelnet que tengan poco contenido o como búsqueda puntual. Un segundo módulo para trabajar con los índices extraídos de NASARI, transformándolos a un formato legible por el servidor que contendrá los datos (JSON).

3.1.3 Etiquetado semántico

Definimos el etiquetado semántico como el proceso de asignación del concepto que mejor se adapta al término o conjunto de términos en un contexto dado. De esta definición tenemos dos aplicaciones directas, la primera sería la obtención de los conceptos representativos de un texto y la segunda sería la obtención del concepto que mejor representa a un término según un contexto dado.

Para la última parte del proyecto, se buscaba un servidor de búsqueda en el que pudiésemos almacenar los datos obtenidos de la extracción de palabras clave y de conceptos para poder trabajar con esos datos. Por la naturaleza de los datos (no estructurados) y la cantidad de los mismos elegimos Elasticsearch (véase sección 2.4).

Las entradas de datos de este servidor son archivos JSON, por lo que tendremos que transformar la salida de los dos primeros módulos para integrarlos en el sistema. Adicionalmente, Elasticsearch usa su propio lenguaje de query (DSL) por lo que configuraremos querys para automatizar el proceso de etiquetado del texto haciendo uso de lo recogido anteriormente.

El resto de querys que se quieran hacer (para tratamiento posterior de los datos más allá del etiquetado) se podrán realizar por consola o desde un navegador al ser un servicio web. Por todo ello, para esta última parte se creará otro módulo para trabajar con las funciones relacionadas con Elasticsearch.

3.2 Codificación, estructura y organización del proyecto

3.2.1 Codificación

Se ha elegido Python como lenguaje de programación por su integración con las diferentes APIs del proyecto (Grampal, babelnet) y por la cobertura de algunas bibliotecas que son especialmente útiles para el proyecto (`Spacy, Elasticsearch, igraph`).

3.2.2 Organización

El proyecto se organiza en corpus de documentos, código fuentes, resultados obtenidos, documentación, y bibliotecas específicas.

El corpus está compuesto por 289 textos extraídos de las memorias anuales en formato plano (.txt).

El código fuente se divide en los siguientes módulos anteriormente comentados:

- Para la parte de extracción de palabras clave dos módulos, uno para trabajar la funcionalidad relacionada con el WS (Web Service) de Grampal y otro con carácter general.
- Para la parte de extracción de conceptos, uno para la conversión de datos de la parte de extracción de conceptos y otro para trabajar directamente con la API de Babelnet.
- Para la parte de etiquetado, un módulo general para el trabajo relacionado con Elasticsearch y otro para la parte de carga de datos.Este último se separó para poder testear y controlar mejor la parte de subida masiva de datos a Elasticsearch (evitando así hacer subidas manuales desde consola).

Los resultados se dividen en dos carpetas `frequencist_list` y `keywords_list` que albergarán el resultado devuelto en función del algoritmo de clasificación utilizado (*frecuencia* o *textrank*).

La documentación contiene un archivo .bat para generar la documentación pero se puede consultar directamente accediendo a la carpeta `build` y luego accediendo al formato deseado, carpeta `pdf` o carpeta `html`.

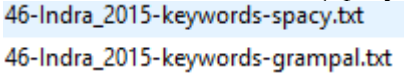
Finalmente en la carpeta de bibliotecas se encuentran los archivos necesarios para la correcta instalación de igraph (tanto en 32 como en 64 bits) evitando errores que arrastra al hacer la instalación de `pip` o de anaconda prompt.

3.2.3 Ficheros

Las salidas de los diferentes módulos generarán ficheros que servirán tanto para una posterior integración con el motor de búsqueda como para tener valor por sí mismos.

Para la primera parte, los ficheros generados se almacenarán en una carpeta cuyo nombre se compondrá de `algoritmo_list`, los ficheros se llamarán de la siguiente manera:

nombre del texto de origen_algoritmo_analizador utilizado (Spacy o Grampal).txt



En la parte de extracción de conceptos ya hemos hablado del archivo de origen, pero teniendo en cuenta que Elasticsearch solo trabaja con archivos JSON definiremos la estructura de la siguiente manera: id/lema/ lista de conceptos (clave, valor):
Traduciendo esto a JSON obtendremos:

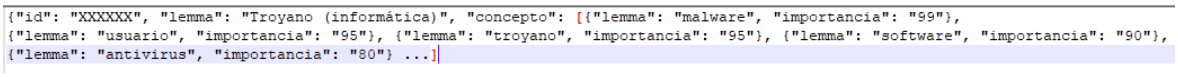


Figura 7: Ejemplo orientativo de estructura del json

Este JSON se añadirá a la carpeta de datos del proyecto.

La parte encargada del etiquetado indexará estos documentos pero no generará archivos adicionales.

3.2.4 Limitaciones

El proyecto tiene varios puntos críticos que limitan su capacidad. Por un lado, la obtención de los textos se realiza manualmente a través de las páginas web de las diferentes empresas, ya que no existe un repositorio común donde obtenerlos.

Adicionalmente, para la parte de extracción de palabras clave se ha implementado específicamente para TextRank pudiendo ser fácilmente ampliado para otros algoritmos basados en grafo, sin embargo no contempla la implementación de otra clase de algoritmos (como pueden ser de *embedding*).

Además, aunque hemos elegido el que a día de hoy constituye el mayor repositorio de relaciones semánticas en español, sigue dejando mucho que desear comparado con los repositorios semánticos en inglés. Esto limita los resultados obtenidos y la calidad de los mismos.

Finalmente, a pesar de haber realizado diferentes tipos de pruebas (caja negra, integración …), los resultados han tenido que ser validados de forma manual al menos al principio del desarrollo. En particular, el listado de exclusión de palabras y algunos detalles más, como retocar configuraciones del compilador de Spacy, han sido realizadas manualmente tras estas validaciones para el refinamiento de los resultados.

4 Desarrollo

4.1 Extracción de palabras clave

Los módulos principales que hacen posible la extracción de palabras clave son keywords.py y ws.py.

4.1.1 Ws.py

En primer lugar para trabajar con las funcionalidades de Grampal se definió una clase Grampal en el fichero ws.py que será posteriormente utilizado por el módulo principal keywords.py.

Esta es la estructura para la clase Grampal desarrollada:

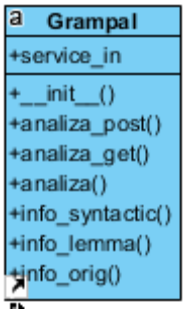


Figura 1: Diagrama de clases de Grampal

Con respecto a la funcionalidad podemos dividirlo en dos tareas, la parte que trabaja con la petición y la respuesta del servicio que hará uso de la librería `requests`, estas funciones serían ***analiza_post()***, ***analiza_get()***, ***analiza()***.

```
def analiza_get(self, phrase):
    if phrase is None:
        print("Error: phrase cannot have a null value")
    return requests.post(self.service_in, data={'texto': phrase})
```

Figura 20: Código de petición al WS de Grampal

El atributo *service_in* es el que almacena la dirección en la que actualmente reside el servicio web de Grampal.

```
12  class Grampal:
13      """``Grampal service class``\n
14      This class implements all the functionality of the Grampal ws
15      """
16      service_in = 'http://leptis.lllf.uam.es/api/grampal/general/'
```

Figura 2: Dirección del WS de Grampal

La otra parte trabajará directamente para extraer los campos de respuesta del objeto devuelto (*Response object*)[24].

Estas funciones son ***info_lemma(text)***, que recoge el lema del token, ***info_orig(text)*** que recoge el token que aparece en el texto, ***info_syntatic(text)*** que recoge la categoría gramatical del token.

[24] Response object doc: https://2.python-requests.org/en/master/api/

4.1.2 keywords.py

A continuación se explican las funciones principales de keywords.py entrando en detalle en las más complicadas (creación de grafo y agrupación de candidatos)

Los argumentos esperados por este módulo son los siguientes:

python keywords.py file co-occurence topnodes service frequency_flag(optional).

```
C:\Users\bug\Desktop\TFG\scripts>python keywords.py 3-BBVA_2014 2 5 spacy
```

Figura 3: Ejemplo de llamada de keywords.py

file: el nombre del archivo sin la extensión
co-occurence: la coocurrencia de los términos (separación) máxima a comprobar
topnodes: top de nodos que se comprobarán para los términos conjuntos
service: servicio (Grampal o Spacy)
frequency_flag: poner un '1' si se desea ordenar por frecuencia (forma parte de los test)

Este módulo consta de una función principal (***main(sys.argv[]***) que hará las llamadas al resto además leerá los datos y escribirá los resultados teniendo en cuenta la organización del proyecto anteriormente comentada.

La primera función relevante que se llama es la de creación del grafo. Están divididas en dos funciones (una para Spacy y otro para Grampal) pero debido a la similitud sólo se comentará una de ellas.

```
def create_graph_spacy(text, k=2):
    """Create a graph with the keywords and their links using spacy as ser
    if text is None:
        print("Error text: The length of the text must be len > 0")
        sys.exit(1)
    if k <= 0:
        print("Error k: The correlation value has to be > 0")
        sys.exit(1)
    graph = igraph.Graph()
    values = []
    values2 = []
    values3 = []
    for sentence in text.sents:
        for token in sentence:
            if token.pos_ in SYNTACTIC_CATEGORY:
                if token.lemma_.upper() not in EXCLUSIONS:
                    values.append((token.lemma_.lower(), token.i))
                    values2.append((token.lemma_.lower(), token.text))
                    values3.append((token.lemma_.lower(), token.pos_))
    uniq = OrderedDict()
    uniq2 = OrderedDict()
    uniq3 = OrderedDict()
    for node in values:
        uniq[node[0]] = node[1]
    for node in values2:
        uniq2[node[0]] = node[1]
    for node in values3:
        uniq3[node[0]] = node[1]
    for key, value in uniq.items():
        graph.add_vertices(1)
        graph.vs[graph.vcount()-1]["lema"] = key
        graph.vs[graph.vcount()-1]["pos"] = value
        graph.vs[graph.vcount()-1]["orig"] = uniq2.get(key)
        graph.vs[graph.vcount()-1]["occur"] = len(value)
        graph.vs[graph.vcount()-1]["fr"] = round(len(value)/len(uniq), 4)
        graph.vs[graph.vcount()-1]["syntactic"] = uniq3.get(key)[0]
    l_aux = list(uniq.keys())
```

Figura 4: Primer bloque de create_graph para Grampal

La función se divide en dos bloques, el primero que crea los nodos y el segundo las aristas.

En primer lugar se segmentan las sentencias y se tokeniza usando las funciones NLP de la biblioteca `Spacy`. Una vez generados los tokens, el atributo *_pos* nos da la categoría gramatical.

Para mejorar los resultados obtenidos se puede aplicar un filtra sintáctico para eliminar aquellos grupos de palabras que no tienen mucho peso semántico, por ejemplo, los artículos. Por ello se comprueba que pertenece a las categorías gramaticales que hemos definido en `SYNTACTIC_CATEGORY`, por defecto nombres y adjetivos, que son aquellas que más peso semántico tienen, para más detalles ir a la parte de resultados).

Si no pertenecen a la lista de palabras excluidas (`EXCLUSIONS`), que son palabras que se repiten en la gran mayoría de textos financieros pero que no aportan mucha información ("año, 2018, 2019, uno, dos" por citar alguno de ellos) se añadirán al grafo.

Finalmente, se inserta en la estructura Graph de la librería `igraph`[25] los datos extraídos de la siguiente forma, una vez que las estructuras auxiliares han formado la lista de lema únicos, crearemos un vértice (***graph.add_vertices(1)***) y rellenaremos los campos que nos interesan:

- *lema*: para el lema
- *pos*: posiciones en las que aparece el lema
- *orig*: palabra de origen del lema (token del texto)
- *occur*: número de apariciones
- *fr*: frecuencia de aparición con respecto a las palabras clave
- *syntactic*: grupo sintáctica

```
167     l_aux = list(uniq.keys())
168     for counter in range(0, len(values)):
169         for i in range(1, k+1):
170             if counter +i < len(values):
171                 current = values[counter][0]
172                 jump = values[counter+i][0]
173                 if current != jump:
174                     try:
175                         if graph.get_eid(l_aux.index(current), l_aux.index(jump))
176                             pass
177                     except:
178                         graph.add_edge(l_aux.index(current), l_aux.index(jump))
179     return graph
```

Figura 5: Segundo bloque de create_graph para Grampal

El segundo bloque, recorre todas las posiciones de los tokens inspeccionados, y mira dentro del rango 1 a K siendo K el coeficiente de co-ocurrencia. Generalmente los valores que dan mejores resultados son valores bajos, 2 o 3, dado que por cómo funciona nuestra lengua cuando usamos un adjetivo sobre algo suele estar cerca de la palabra a la que hace referencia.
Podemos ver un pequeño ejemplo del coeficiente de co-ocurrencia en la siguiente frase: "El algoritmo es rápido pero aun así es mejorable en varios aspectos"

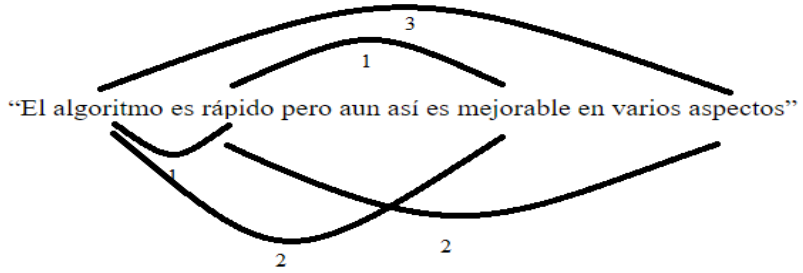


Figura 6: Ejemplo de la co-ocurrencia en una sentencia

[25] Doc igraph https://igraph.org/python/#docs

Finalmente, si ya existe la conexión (***graph.get_eid***) pasamos y si no la creamos (***graph.add_edge***).

Con el grafo generado el resto de funciones son bastante triviales. Destacamos entre ellas *pagerank* que asigna un nuevo atributo "*value*" a nuestro grafo, devolviendo la lista de valores ordenados después de aplicar el algoritmo.

```
    values = graph.pagerank()
    for counter in range(0, len(values)):
        graph.vs[counter]["value"] = round(values[counter], 4)
    return values
```

Figura 7: Función de evaluación Pagerank

Por último, se añade una función que une los nodos que mejor puntuación han obtenido al aplicarle el algoritmo de evaluación, de la misma forma que para la creación del grafo se divide en dos funciones,

concatenate_candidates_(Grampal o Spacy)

Su funcionamiento es sencillo, se itera para los *x* primeros nodos del grafo. En cada iteración se comprueba si alguna de las apariciones del nodo en el texto es adyacente a algún nodo comprendido entre los *x* primeros. Esta idea que se expone en el texto de TextRank: Bringing Order into Texts pero se ha llevado un paso más allá comprobando la aparición de conectores lógicos.

Gracias a esta mejora, palabras claves conjuntas ("*multiwords*") (como se define en el texto) del estilo de "fábrica de acero" podrán aparecer al considerar que el conector al no tener significado por sí mismo no cambia el significado del conjunto formado.

```
351     if graph is None:
352         print("Error graph: Empty graph")
353         return None
354     if nodes is None:
355         print("Error nodes: Top nodes is empty")
356         return None
357     if text is None:
358         print("Error text: The length of the text must be len > 0")
359         return None
360     ginstance = Grampal()
361     pos = 0
362     control = 0
363     multiword = OrderedDict()
364
365     for sentence in text.sents:
366         response = ginstance.analiza(sentence.text)
367         if response.status_code != 200:
368             continue
369         lines = response.text.splitlines()
370         for i in range(len(lines)):
371             control = 0
372             if lines[i] != "":
373                 if i < len(lines)-1:
374                     for node in nodes:
375                         if pos in graph.vs[node]["pos"]:
376                             if control == 1:
377                                 break
378                             for node2 in nodes:
379                                 if pos+1 in graph.vs[node2]["pos"]:
380                                     lema_aux = graph.vs[node]["lema"]+"_"+graph.vs[node2]["lema"]
381                                     multiword[lema_aux] = graph.vs[node]["value"]+graph.vs[node2]["value"]
382                                     multiword[lema_aux] = graph.vs[node]["occur"]+graph.vs[node2]["occur"]
383                                     multiword[lema_aux] = graph.vs[node]["fr"]+graph.vs[node2]["fr"]
384                                     multiword[lema_aux] = (graph.vs[node]["orig"], graph.vs[node2]["orig"])
385                                     multiword[lema_aux] = (graph.vs[node]["syntactic"], graph.vs[node2]["syntactic"])
386                                     multiword[lema_aux] = pos
387                                     control = 1
388                                 break
389                 if lines[i+1] != "" and control == 0:
390                     if ginstance.info_lemma(lines[i+1]).lower() in CONNECTORS:
391                         if i < len(lines)-2:
392                             for node2 in nodes:
393                                 if pos+2 in graph.vs[node2]["pos"]:
394                                     lema_aux = graph.vs[node]["lema"]+"_"+ginstance.info_lemma(lines[i+1]).low
```

Figura 8: Extracto de función de creación de palabras clave múltiple

Adicionalmente aunque no estaba contemplado en las primeras fases del diseño se ha añadido más funcionalidad. Entre las cuales están una función de dibujo de grafos `print_graph(graph, path)` y un conjunto de configuraciones que afinan los resultados:

```
453     nlp.tokenizer = custom_tokenizer(nlp)
454     special_case = [{ORTH: '•', LEMMA: '•', POS: 'PUNCT'}]
455     nlp.tokenizer.add_special_case('•', special_case)
456     special_case2 = [{ORTH: '—', LEMMA: '—', POS: 'PUNCT'}]
457     nlp.tokenizer.add_special_case('—', special_case2)
```

Figura 9: Extracto de Creación caso especial en el tokenizer

El tokenizer que viene por defecto en spacy admite la adición de casos especiales que serán tenidos en cuentas antes de pasarle el documento a procesar.

A través de la función ***add_special_case*** contenida en `nlp.tokenizer` se puede cambiar la respuesta para determinados lemas. En la Figura 20 por ejemplo se han añadido símbolos de puntuación que el analizador no detectaba.
Estos errores ocasionan errores a la hora de segmentar las sentencias.

Adicionalmente, es posible retocar la función del compilado del tokenizer de Spacy, editando las reglas que sigue para su análisis léxico, podemos evitar la separación de tokens que estén separados por caracteres en específico.

En la Figura 21 por ejemplo se ha añadido reglas para evitar la separación del carácter '-' por lo tanto en Mato García-Ansorena se formarían solo 2 tokens y no 4.

```
def custom_tokenizer(nlp):
    if nlp is None:
        print("Error nlp: nlp can't be null ")
        return None
    prefix_re = re.compile(r'''^[—•\["']''')
    suffix_re = re.compile(r'''[—•,.\]\)"']$''')
    infix_re = re.compile(r'''[•~]''')
    simple_url_re = re.compile(r'''^https?://''')
    return Tokenizer(nlp.vocab, prefix_search=prefix_re.search,
```

Figura 29: Configuraciones de compilado del tokenizer

Finalmente, se volcarán los resultados en un archivo.txt.

La estructura interna de los archivos generados será la siguiente:

- Una línea por palabra clave (individual o compuesta), con una línea en blanco para separar los candidatos compuestos de los individuales.

Los campos estarán separados por tabulaciones y serán los siguientes:

- Lema
- Valor generado por el algoritmo
- Número de apariciones
- Frecuencia de aparición (con respecto al total de palabras clave)
- Listado de Formas (Origen del lema en el texto)
- Categoría gramatical

Adicionalmente, se establece un campo adicionalmente en la parte de candidatos compuestos con la posición en la que aparece en el texto.

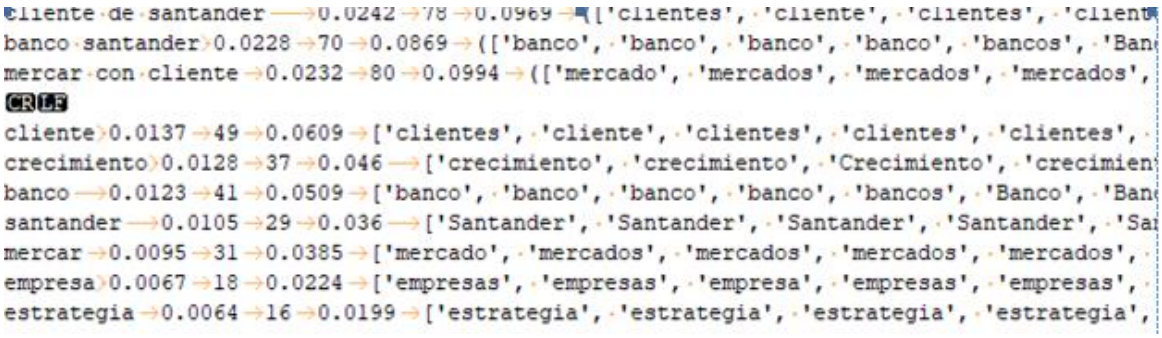


Figura 10: Ejemplo de fichero generado en keywords.py

4.2 Extracción de conceptos

En esta parte del desarrollo se había diseñado una serie de funciones para trabajar con los conceptos y sentidos de BabelNet mediante peticiones al servicio *online*. Desafortunadamente, nos vimos limitados por el número de peticiones diarias que admite BabelNet (1000). Así, en un primer lugar se desarrollaron dos módulos para trabajar con la API de Babelnet (`Concepts/Senses.py`), posteriormente cuando nos facilitaron los índices de NASARI simplemente hicimos un módulo de conversión de datos.

Comentaremos brevemente estos módulos, aunque su uso ha sido más a modo de pruebas, podría servir para futuras implementaciones si se eliminase la restricción de uso.
Las dos operan como un WS alojado en 'https://babelnet.io/v5/'[26] .

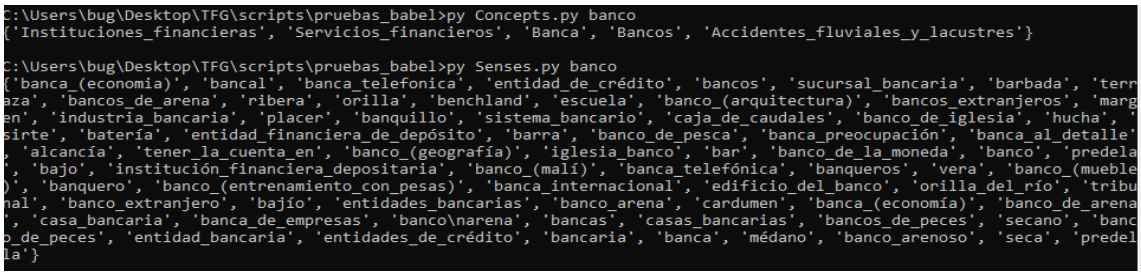


Figura 11: Ejemplo de respuesta del WS de BabelNet

`Senses.py` utiliza el metodo `GetSenses()` para trabajar con los sentidos de una palabra dada y `Concepts.py` con los métodos `getSynsetIds()` y `getSynset()` para trabajar con los conceptos .

El Módulo de conversión (`create_json.py`) será el encargado de hacer la transformación de la información de NASARI para que sea legible por nuestro motor de búsquedas (Elasticsearch).

Usando como referencia los mapeos de Elasticsearch[27] se crea la siguiente estructura:

[26] Documentación API BabelNet https://babelnet.org/guide

[27] Elastic_mapping: https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html

```
def single_json(file_name):
    directory = os.path.join(os.path.dirname(os.getcwd()), "data", "json", "bulk"+".json")
    with open(file_name, encoding="utf-8") as file:
        with open(directory, "w+", encoding="utf-8") as outfile:
            for line in file:
                aux = re.sub(r'[^\w+-.\t]', '', line)
                dic = {}
                row = aux.split("\t")
                dic["id"] = row[0]
                dic["lemma"] = row[1]
                dic["concepto"] = []
                for tupla in row[2:]:
                    conceptos = {}
                    concept = tupla.split("_")
                    conceptos["lemma"] = concept[0]
                    conceptos["importancia"] = concept[1]
                    dic["concepto"].append(conceptos)
                json.dump(dic, outfile, ensure_ascii=False)
                print("", file=outfile)
```

Figura 12: Función de creación de archivo JSON

Se hace una normalización de los términos del repositorio utilizando la funcionalidad proporcionada por el módulo `Python RegEx(re)`.
A continuación, se agrupan en un diccionario (dic) los campos requeridos y finalmente se hace un volcado del proceso generado.

La llamada a este módulo se realiza con los siguientes argumentos:
`python create json.py modo origen de datos`

```
C:\Users\bug\Desktop\TFG\scripts>py create_json.py 0 NASARI_lexical_spanish
```

Figura 13: Ejemplo de llamada de create_json.py

Modo puede valer 0 para crear un solo archivo JSON y 1 para crear un JSON por línea.

Por defecto, se encuentra un documento llamado NASARI_lexical_spanish.txt en la carpeta data del proyecto que servirá como parámetro de entrada del módulo, la salida será a la carpeta json dentro de data, el archivo se llama bulk.json .

```
1 {"id": "bn00000002n", "lemma": "La Haya",
2 "concepto": [{"lemma": "bajos", "importancia": "243.44"},
3 {"lemma": "neerlandés", "importancia": "211.82"},
4 {"lemma": "varsovia", "importancia": "201.99"},
5 {"lemma": "países", "importancia": "190.76"},
6 {"lemma": "leiden", "importancia": "164.71"},
7 {"lemma": "ciudad", "importancia": "132.62"},
8 {"lemma": "holanda", "importancia": "97.56"}, {"lemma": "]
9
```

Figura 14: Ejemplo de archivo JSON generado

4.3 Etiquetado semántico

Antes de pasar a la explicación de los módulos implantados, hace falta instalar y configurar el motor de búsqueda. La información relativa a estos dos pasos se puede encontrar en el manual de instalación de la sección de Anexos y en la página principal de Elasticsearch[28].

[28] ES: https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started-install.htm

No entraremos en detalle sobre la API pero sí que comentaremos las funciones principales para trabajar con el motor de búsqueda. Trabajamos con la biblioteca `curl` para la llamada al servidor de Elasticsearch (por defecto se crea en el puerto 9200 y los ejemplo así lo han tenido en cuenta).

En Elasticsearch, los datos se agrupan en índices (véase Sección 2.4). Así que para ver el estado de nuestro servidor y la cantidad de datos indexados entre otros datos, tenemos que hacerlo mediante llamadas al servidor de Elasticsearch. Para facilitar esta tarea, utilizamos el comando `curl`[29] que facilita la comunicación con un servidor (en nuestro caso el de Elasticsearch.

El siguiente comando muestra los índices de nuestro motor:

```
C:\Users\bug\Desktop\TFG\scripts>curl -X GET "localhost:9200/_cat/indices?v"
health status index uuid pri rep docs.count docs.deleted store.size pri.store.size
```

Figura 15: Comando GET indices

Para borrado de índices:

```
C:\Users\bug\Desktop\TFG\scripts>curl -X DELETE "localhost:9200/keywords?pretty"
```

Figura 16: Comando DELETE índice

Para la creación de índices:

```
C:\Users\bug\Desktop\TFG\scripts>curl -X PUT "localhost:9200/customer?pretty"
{
  "acknowledged" : true,
  "shards_acknowledged" : true,
  "index" : "customer"
}
```

Figura 17: Comando PUT indice

Para la parte del etiquetado, hemos realizado la subida de toda la red de información contenida en NASARI, que hemos transformado gracias al módulo anterior a un formato JSON legible por nuestro motor de búsqueda.

Esta tarea la realiza el módulo `elastic_bulk`, a través de la API de Elasticsearch usando la función `bulk` para cargar los archivos en el sistema.

```
13  def decode_nginx_log(_nginx_fd):
14      """!@brief  Function that parse the source information from a json.
17      with open(_nginx_fd,encoding='utf-8') as f:
18          for each_line in f:
19              yield json.loads(each_line)
20
21  def es_add_bulk(nginx_file):
22      """!@brief  Function that bulk the information from a json.
25      es = Elasticsearch(hosts = [{'host': 'localhost', 'port': 9200}])
26      k = ({
27          "_index": "concepts",
28          "_type": "doc",
29          "_source": registro,
30          } for registro in decode_nginx_log(nginx_file))
31      helpers.bulk(es, k)
```

Figura 18: Función bulk

```
python elastic_bulk.py file_name
```

[29] Curl: https://curl.haxx.se/

```
C:\Users\bug\Desktop\TFG\scripts>python elastic_bulk.py bulk

C:\Users\bug\Desktop\TFG\scripts>curl -X GET "localhost:9200/_cat/indices?v"
health status index    uuid                   pri rep docs.count docs.deleted store.size pri.store.size
yellow open   concepts ALmSSKY_S1yk10hqlOvhmg   1   1    1051438            0        3gb            3gb
```

Figura 19: Prueba de indexado de los documentos

Nota: Este módulo crea por defecto el índice "concepts", no hace falta crearlo con el comando PUT.

Una vez que los datos están indexados podremos utilizar las diferentes funciones del módulo `elastic.py` para el etiquetado semántico de los textos tratados en la primera parte. El módulo permite dos funcionalidades: etiquetar el texto con los conceptos más representativos y transformar la lista de palabras clave en una lista de conceptos.

`py elastic.py nombre_archivo modo modelo alcance`
Los parámetros de ejecución son los siguientes:

- *nombre_archivo*: el nombre del archivo que contiene las palabras claves recogidas de la parte uno.
- *modo*: Modo de acceso a los datos del algoritmo (0,1).
- *modelo*: Modelo de incremento del algoritmo (0,1).
- *alcance*: Número de palabras clave a tener en cuenta.

```
C:\Users\bug\Desktop\TFG\scripts>py elastic.py 1-BancoSantander_2014-keywords-grampal 0 0 50
```

Figura 20: Ejemplo de llamada de elastic.py

A continuación se explican las funciones principales de este módulo, la primera relativa a la extracción de conceptos.
`concept_extraction(es,list_keywords,mode,model,reach)`

```
16  def concept_extraction(es,list_keywords,mode,model,reach=10000):
17      query={}
18      bool={}
19      should={}
20      counter=0
21      array_lemma=[]
22      top30=list_keywords[29][1]
23      for key,value in list_keywords:
24          if counter < reach:
25              counter+=1
26              aux_query={}
27              aux_query["query"]=key
28              if model ==0:
29                  increase=value
30              else:
31                  increase=((value*(1/top30))**1.1)
32              aux_query["boost"]= 1+increase
33              array_lemma.append(aux_query)
34      array_match=[]
35      for aux in array_lemma:
36          lemma={}
37          if mode ==0:
38              lemma["lemma"]=aux
39          elif mode ==1:
40              lemma["concepto.lemma"]=aux
41          array_match.append(lemma)
42      arrray_should=[]
43      for aux2 in array_match:
44          match={}
45          match["match_phrase"]=aux2
46          arrray_should.append(match)
47      should["should"]=arrray_should
48      bool["bool"]=should
49      query["query"]=bool
50      return  es.search(index='concepts',body=query,size= "20")
51
```

Figura 21: Cuerpo de la función de extracción de conceptos

La función recibe de 5 argumentos de entrada, que son los siguientes:

- *es*: el servidor de Elasticsearch.
- *list_keywords*: listado de tuplas de (clave,valor) generado en la primera parte.
- *mode*: modo que puede ser 0 si se quiere hacer una búsqueda por lemas o 1 si se quiere hacer una búsqueda por la aparición de la palabra buscada como concepto de otros lemas (son ideas parecidas pero esta última toma muchos más datos arrojando diferentes resultados).
- *model*: Modelo seguido para el incremento del atributo boost[30](ponderación) de Elasticsearch.

A la hora de aplicar la query (consulta) con el lenguaje DSL se puede introducir una ponderación de los campos, es decir dar más importancia a algunos campos en específico. Esta idea la hemos junto con el resultado devuelto en el algoritmo de Pagerank, dando más importancia en la búsqueda de conceptos a aquellos términos que aparecen en los primeros puestos en el algoritmo.

Teniendo en cuenta que la base del boost es "1.00", hemos seguido dos modelos. El primero un modelo con un incremento lineal sumándole el valor obtenido a la hora de ejecutar Pagerank. Debido a la cantidad de términos y al funcionamiento interno del atributo Boost, los cambios apenas eran apreciables. En la mayoría de textos no implicaba una variación directa de los conceptos más representativos aunque los valores de los mismos variaban ligeramente.

En respuesta a esta problemática se implantó un modelo exponencial que incrementa el valor para los 30 primeros términos y disminuye el valor para los siguientes. Arrojando unos resultados más apreciables al anular prácticamente los últimos valores de Pagerank y incrementar notablemente el valor de los primeros.

```
top30=list_keywords[29][1]
for key,value in list_keywords:
    if counter < reach:
        counter+=1
        aux_query={}
        aux_query["query"]=key
        if model ==0:
            increase=value
        else:
            increase=((value*(1/top30))**1.1)
        aux_query["boost"]= 1+increase
        array_lemma.append(aux_query)
array_match=[]
```

Figura 22: Detalle del incremento implantado

Por último, tendríamos el alcance que es número de palabras clave que se quieren tomar en cuenta.

El resto del algoritmo solo apoya la construcción de la que será la encarga de comunicarse con Elasticsearch a través de una llamada a search que ejecutará el cuerpo de la consulta en el índice que habíamos creado para nuestra red semántica.

```
    return  es.search(index='concepts',body=query,size= "20")
```

[30] ES boost: https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-boost.html

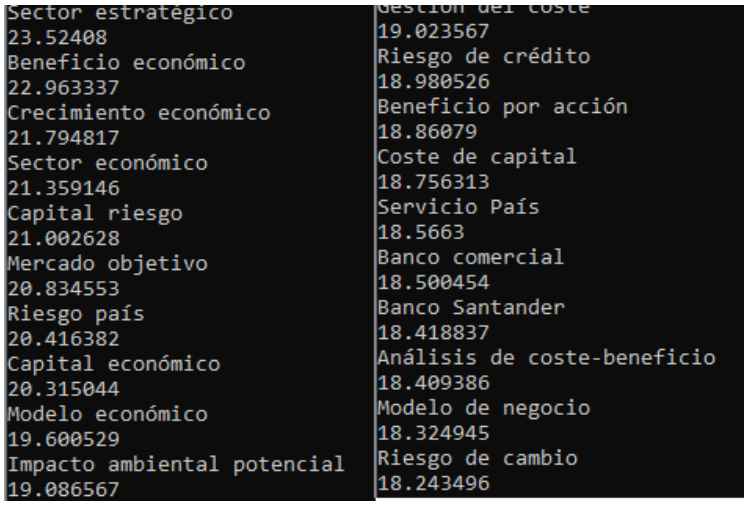


Figura 23: Resultado de la búsqueda de conceptos en el texto 1-BancoSantander-Grampal.txt

Ejemplo de resultados para el comando:
`py elastic.py 1-BancoSantander_2014-keywords-grampal 0 0 50`

La otra función destacable será la encargada de crear el listado de palabras clave y su asociación de conceptos.

```
extract_list(es,list_keywords,mode,model,reach)
```

```
def extract_list(es,list_keywords,mode,model,reach=10000):
    query={}
    match={}
    lemma={}
    counter=0
    query_aux={}
    aux=0
    reference=0
    reference_original=0
    list={}

    results=concept_extraction(es,list_keywords,mode,model,reach)
    for data in results["hits"]["hits"]:
        reference_original +=data['_score']
    reference=reference_original
    for a,b in list_keywords:
        if counter < reach:
            query_aux["query"]=a
            query_aux["boost"]=1
            list[a]=a
            lemma={}
            if mode == 0:
                lemma["lemma"]=query_aux
            elif mode ==1:
                lemma["concepto.lemma"]=aux
            match["match"]=lemma
            query["query"]=match
            results= es.search(index='concepts',body=query,size= "10")
            for data in results["hits"]["hits"]:
                list_keywords[counter]=data['_source']['lemma'],b
                results2=concept_extraction(es,list_keywords,mode,model,reach)
                aux=0
                for data2 in results2["hits"]["hits"]:
                    aux +=data2['_score']
                    if(aux > reference):
                        reference=aux
                        list[a]=data['_source']['lemma']
            reference=reference_original
            list_keywords[counter]=list[a],b
            counter+=1
    return list
```

Figura 24: Cuerpo de la función de extracción de conceptos

La idea general del algoritmo es sencilla, buscamos para cada palabra clave sus diez primeros conceptos, iteramos el algoritmo de extracción de conceptos, reemplazando en la lista *list_keywords* el lema por uno de estos conceptos. Como las búsquedas tienen asociado una puntuación, podemos compararlas, quedándonos con el concepto que mejor se ajuste al resto.

Las otras funciones del módulo solo sirven para la inserción de los datos, como es el caso de `insert_concept_results()` e `insert_list_results()`. Además de una función de búsqueda, que permite recuperar las búsquedas que ya han sido indexadas previamente en el documento.

```
C:\Users\bug\Desktop\TFG\scripts>py elastic.py 1-BancoSantander_2014-keywords-grampal 0 0 50
Busqueda previa detectada
Mostrando conceptos
[{'lemma': 'Sector estratégico'}, {'lemma': 'Beneficio económico'}, {'lemma': 'Crecimiento económico'}, {'lemma': 'Secto
' económico'}, {'lemma': 'Capital riesgo'}, {'lemma': 'Mercado objetivo'}, {'lemma': 'Riesgo país'}, {'lemma': 'Capital
económico'}, {'lemma': 'Modelo económico'}, {'lemma': 'Impacto ambiental potencial'}, {'lemma': 'Gestión del coste'}, {'
lemma': 'Riesgo de crédito'}, {'lemma': 'Beneficio por acción'}, {'lemma': 'Coste de capital'}, {'lemma': 'Servicio País
'}, {'lemma': 'Banco comercial'}, {'lemma': 'Banco Santander'}, {'lemma': 'Análisis de coste-beneficio'}, {'lemma': 'Mod
elo de negocio'}, {'lemma': 'Riesgo de cambio'}]
Mostrando listado palabras clave / concepto
[{'keyword': 'CLIENTE', 'concept': 'Cliente liviano'}, {'keyword': 'BANCO', 'concept': 'Banco comercial'}, {'keyword': '
CRECIMIENTO', 'concept': 'Crecimiento económico'}, {'keyword': 'EURO', 'concept': 'AFC Euro Kickers'}, {'keyword': 'GRUP
O', 'concept': 'Grupo Unicomer'}, {'keyword': 'BENEFICIO', 'concept': 'Beneficio por acción'}, {'keyword': 'CAPITAL', 'c
oncept': 'Capital riesgo'}, {'keyword': 'MERCADO', 'concept': 'Mercado extrabursátil'}, {'keyword': 'RIESGO', 'concept':
 'Capital riesgo'}, {'keyword': 'OBJETIVO', 'concept': 'Objetivo varifocal'}, {'keyword': 'MAYOR', 'concept': 'Semieje m
```

Figura 25: Ejemplo de respuesta de la función search

5 Integración, pruebas y resultados

A continuación, se exponen las diferentes pruebas realizadas a lo largo del proyecto.

Estas se componen principalmente de pruebas de caja negra y pruebas de integración.

Adicionalmente, hay que destacar que a lo largo de todo se han realizado pruebas de validación en las que se han ido depurando los resultados generados. Todo este proceso no se ha documentado al ser un trabajo diario en el desarrollo del proyecto.

Las pruebas de caja negra las realiza un módulo pruebas.py que hace llamadas a toda la funcionalidad del proyecto. El alcance de estas pruebas es limitado, debido al hecho que los resultados finales son archivos en texto plano que tienen que ser valorados manualmente.

Por ello para reforzarlas se han ejecutado pruebas de integración sobre las funciones principales.

5.1 Pruebas de caja negra

Las pruebas de caja negra es una técnica de pruebas del software en el que comprobaremos la funcionalidad sin tener en cuenta la estructura de la función. Generalmente se crea un conjunto de entradas que ejercitan todos los requisitos funcionales del programa.

Para estas pruebas se ha hecho uso de la librería estándar de python `unittest`.

	MODULO	MÉTODO	DESCRIPCION
2	TestGrampal	analiza	Función encargada de enviar la petición al servicio Grampal y devolver su respuesta
3		Info_syntactic	Función encargada de recuperar la información sintáctica de la respuesta del servicio
4		Info_lemma	Función encargada de recuperar la información del lema de la respuesta del servicio
5		Info_orig	Función encargada de recuperar la información del token de origen de la respuesta del servicio
6	TestKeywords	create_graph_grampal	Función encargada de crear el grafo usando el analizador síntactico de Grampal
7		create_graph_spacy	Función encargada de crear el grafo usando el analizador síntactico de Spacy
8		pagerank	Función de evaluación basada en pagerank
9		topnodes	Función de extracción de los nodos con mejores valoraciones(top)
10		sort_values	Función de evaluación basada en el valor dado por el algoritmo aplicado
11		sort_occurences	Función de evaluación basada en la frecuencia de aparción del lema
12		concatenate_candidates_spacy	Función de búsqueda y creación de tokens compuestos basada en spacy
13		concatenate_candidates_grampal	Función de búsqueda y creación de tokens compuestos basda en grampal
14		custom_tokenizer	Función de parametrización del compilador nlp de spacy, añadiendo excepciones a la hora del
15	TestCreate_json	single_json	Función de conversión a un json a partir de los indices de BabelNet
16		multiple_json	Función de conversión a varios json(uno por objeto) a partir de los indices de BabelNet
17	Testelastic_json	concept_extraction	Funcion de extracción de conceptos
18		extract_list	Funcion de extracción de tupla(keyword/concept)
19		search_results	Función de búsqueda de documentos indexados
20		insert_concept_results	Función de indexación de documentos
21		insert_list_results	Funcion de indexación de documentos

Figura 1: Resumen de los test de caja negra

Figura 2: Extracto de la ejecución de las pruebas de Caja negra

5.2 Pruebas de integración

Las pruebas de integración están diseñadas para comprobar el correcto funcionamiento de los módulos desarrollados y la integración de los diferentes módulos que formarán el proyecto. Para llevarlas a cabo se ha ejecutado una batería de pruebas que abarca una mayoría de las casuísticas posibles.

5.2.1 Extracción de palabras claves

Para comprobar las funciones de este módulo se han recopilado los resultados sobre todo el conjunto de textos contenidos en la carpeta data.

Se han variado los parámetros de entrada de los algoritmos a probar. Entre los atributos testeados están el filtrado sintáctico (Atributo `SYNTACTIC_GROUP`), el coeficiente de co-occurencia y el número de palabras clave a comprobar para la extracción de las keywords.

En etapas tempranas del desarrollo con estas pruebas se detectaron numerosos errores.

Por poner un ejemplo, sobre las funciones que encontraban palabras clave compuestas se detectó que la librería Spacy trataba su atributo de posición del token de una forma diferente a lo que establecimos en Spacy. Teniendo por consecuencia la creación de claves inexistentes y en el peor de los casos, que aparezcan caracteres no contemplados en los otros módulos provocando fallos en la ejecución.

1	1-BancoSantander_2014	python keywords.py 1-BancoSantander_2014 2 5 spac	python keywords.py 1-BancoSantander_2014 3 5 spacy
2	1-BancoSantander_2015	python keywords.py 1-BancoSantander_2015 2 5 spac	python keywords.py 1-BancoSantander_2015 3 5 spacy
3	1-BancoSantander_2016	python keywords.py 1-BancoSantander_2016 2 5 spac	python keywords.py 1-BancoSantander_2016 3 5 spacy
4	1-BancoSantander_2017	python keywords.py 1-BancoSantander_2017 2 5 spac	python keywords.py 1-BancoSantander_2017 3 5 spacy
5	2-Inditext_2014	python keywords.py 2-Inditext_2014 2 5 spacy	python keywords.py 2-Inditext_2014 3 5 spacy
6	2-Inditext_2015	python keywords.py 2-Inditext_2015 2 5 spacy	python keywords.py 2-Inditext_2015 3 5 spacy
7	2-Inditex_2016	python keywords.py 2-Inditex_2016 2 5 spacy	python keywords.py 2-Inditex_2016 3 5 spacy
8	3-BBVA_2014	python keywords.py 3-BBVA_2014 2 5 spacy	python keywords.py 3-BBVA_2014 3 5 spacy
9	3-BBVA_2015	python keywords.py 3-BBVA_2015 2 5 spacy	python keywords.py 3-BBVA_2015 3 5 spacy
10	3-BBVA_2016	python keywords.py 3-BBVA_2016 2 5 spacy	python keywords.py 3-BBVA_2016 3 5 spacy
11	3-BBVA_2017	python keywords.py 3-BBVA_2017 2 5 spacy	python keywords.py 3-BBVA_2017 3 5 spacy
12	4-Telefonica_2014	python keywords.py 4-Telefonica_2014 2 5 spacy	python keywords.py 4-Telefonica_2014 3 5 spacy
13	4-Telefonica_2015	python keywords.py 4-Telefonica_2015 2 5 spacy	python keywords.py 4-Telefonica_2015 3 5 spacy
14	4-Telefonica_2016	python keywords.py 4-Telefonica_2016 2 5 spacy	python keywords.py 4-Telefonica_2016 3 5 spacy

Figura 3: Extracto de la ejecución de las pruebas de integración

Como ya hemos comentado, se han hecho pruebas de validación para el formato de creación del documento, siendo aprobado por los encargados del departamento de lingüística de la UAM.

5.2.2 Extracción de conceptos

Para la extracción de conceptos se creó en un primer lugar una función para crear elementos JSON unitarios (esta función es `single_json` del módulo `create_json.py`). A continuación estos documentos se indexaron con la función ***index***[31] del módulo de elastic.py.

[31] index https://elasticsearch-py.readthedocs.io/en/master/api.html#indices

```
    res = es.index(index='keywords',doc_type='doc',id=1,body=concept)
```

Con estas pruebas se pudo mejorar la estructura del JSON implementada, entendiendo el detalle de la indexación de documentos en elasticsearch.

Estas pruebas se pueden replicar mediante la llamada con los parámetros adecuados al módulo create_json.py. Para chequeos adicionales se puede comprobar los logs de la respuesta de la función index.

Adicionalmente para validar la estructura JSON se ha utilizado la herramienta https://jsonlint.com/ que es un validador de JSON.

5.2.3 Etiquetado

Para las pruebas de etiquetado, se probó en primer lugar mediante query DSL la creación de índices. Haciendo subidas individuales de los archivos a través de la consola de datos, y probando diferentes consultas que nos podrían resultar útiles para poder recabar la información que nos interesaba.

Posteriormente una vez entendido el funcionamiento del motor de búsqueda se buscó una de las api de elasticsearch en python para proceder con su integración.

5.3 Resultados

En este apartado presentamos los resultados obtenidos en los diferentes procesos del proyecto, adicionalmente mostraremos aquellas configuraciones que mejores resultados han proporcionado.

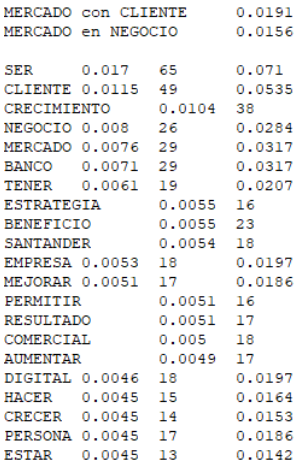
5.3.1 Extracción de palabras clave

En una primera evaluación de los resultados se probaron diferentes filtros sintácticos.

A continuación, se muestran los resultados obtenidos para estos filtros:

Figura 4: Resultados con filtrado sintáctico

También se probó juntando las tres categorías gramaticales

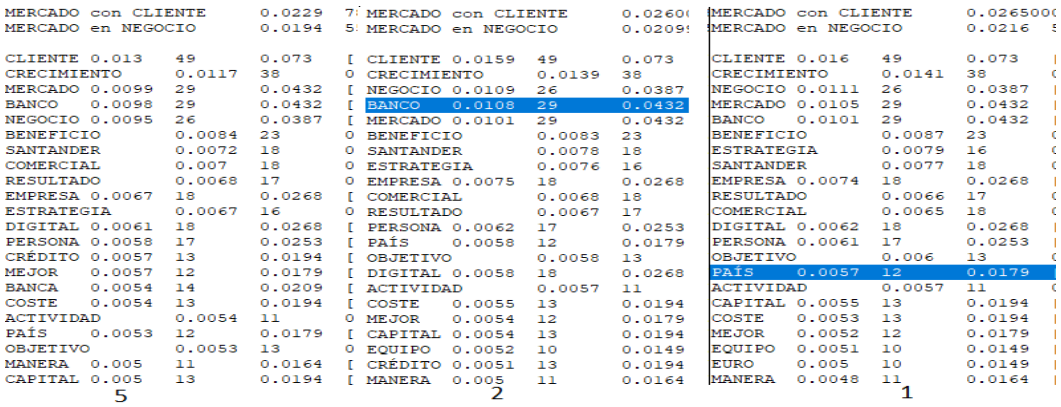


MERCADO con CLIENTE			0.0191
MERCADO en NEGOCIO			0.0156
SER	0.017	65	0.071
CLIENTE	0.0115	49	0.0535
CRECIMIENTO		0.0104	38
NEGOCIO	0.008	26	0.0284
MERCADO	0.0076	29	0.0317
BANCO	0.0071	29	0.0317
TENER	0.0061	19	0.0207
ESTRATEGIA		0.0055	16
BENEFICIO		0.0055	23
SANTANDER		0.0054	18
EMPRESA	0.0053	18	0.0197
MEJORAR	0.0051	17	0.0186
PERMITIR		0.0051	16
RESULTADO		0.0051	17
COMERCIAL		0.005	18
AUMENTAR		0.0049	17
DIGITAL	0.0046	18	0.0197
HACER	0.0045	15	0.0164
CRECER	0.0045	14	0.0153
PERSONA	0.0045	17	0.0186
ESTAR	0.0045	13	0.0142

Figura 5: Resultados con filtrado sintáctico ADJETIVO/VERBO/NOMBRE

La riqueza de los datos y sobretodo la aparición de palabras claves compuesta son mejores haciendo un filtro que solo tiene en cuenta adjetivos y sustantivos. Los resultados tienen este filtro (Adjetivo, nombre) por defecto.

A continuación, se prueba los resultados de la variación de la co-occurencia.



MERCADO con CLIENTE			0.0229	7	MERCADO con CLIENTE			0.0260	MERCADO con CLIENTE			0.026500
MERCADO en NEGOCIO			0.0194	5	MERCADO en NEGOCIO			0.0209	MERCADO en NEGOCIO			0.0216
CLIENTE	0.013	49	0.073	[	CLIENTE	0.0159	49	0.073	CLIENTE	0.016	49	0.073
CRECIMIENTO		0.0117	38	0	CRECIMIENTO		0.0139	38	CRECIMIENTO		0.0141	38
MERCADO	0.0099	29	0.0432	[	NEGOCIO	0.0109	26	0.0387	NEGOCIO	0.0111	26	0.0387
BANCO	0.0098	29	0.0432	[	BANCO	0.0108	29	0.0432	MERCADO	0.0105	29	0.0432
NEGOCIO	0.0095	26	0.0387	[	MERCADO	0.0101	29	0.0432	BANCO	0.0101	29	0.0432
BENEFICIO		0.0084	23	0	BENEFICIO		0.0083	23	BENEFICIO		0.0087	23
SANTANDER		0.0072	18	0	SANTANDER		0.0078	18	ESTRATEGIA		0.0079	16
COMERCIAL		0.007	18	0	ESTRATEGIA		0.0076	16	SANTANDER		0.0077	18
RESULTADO		0.0068	17	0	EMPRESA	0.0075	18	0.0268	EMPRESA	0.0074	18	0.0268
EMPRESA	0.0067	18	0.0268	[	COMERCIAL		0.0068	18	RESULTADO		0.0066	17
ESTRATEGIA		0.0067	16	0	RESULTADO		0.0067	17	COMERCIAL		0.0065	18
DIGITAL	0.0061	18	0.0268	[	PERSONA	0.0062	17	0.0253	DIGITAL	0.0062	18	0.0268
PERSONA	0.0058	17	0.0253	[	PAÍS	0.0058	12	0.0179	PERSONA	0.0061	17	0.0253
CRÉDITO	0.0057	13	0.0194	[	OBJETIVO		0.0058	13	OBJETIVO		0.006	13
MEJOR	0.0057	12	0.0179	[	DIGITAL	0.0058	18	0.0268	PAÍS	0.0057	12	0.0179
BANCA	0.0054	14	0.0209	[	ACTIVIDAD		0.0057	11	ACTIVIDAD		0.0057	11
COSTE	0.0054	13	0.0194	[	COSTE	0.0055	13	0.0194	CAPITAL	0.0055	13	0.0194
ACTIVIDAD		0.0054	11	0	MEJOR	0.0054	12	0.0179	COSTE	0.0053	13	0.0194
PAÍS	0.0053	12	0.0179	[	CAPITAL	0.0054	13	0.0194	MEJOR	0.0052	12	0.0179
OBJETIVO		0.0053	13	0	EQUIPO	0.0052	10	0.0149	EQUIPO	0.0051	10	0.0149
MANERA	0.005	11	0.0164	[	CRÉDITO	0.0051	13	0.0194	EURO	0.005	10	0.0149
CAPITAL	0.005	13	0.0194	[	MANERA	0.005	11	0.0164	MANERA	0.0048	11	0.0164
5					2				1			

Figura 6: Resultados con variación de co-occurencia

Los cambios no son muy apreciables pero los tiempos de ejecución al aumentar este valor se multiplican, para mantener un punto intermedio entre coherencia y tiempos se decide dejar en 2.

Para la creación de palabras duplicados estudiamos la variación del “Top” (valores más altos)

El aumentar este valor produce un exceso de claves conjuntas, esto afecta negativamente a los resultados de consulta de elasticsearch, por lo tanto, se decide mantener un valor de 2.

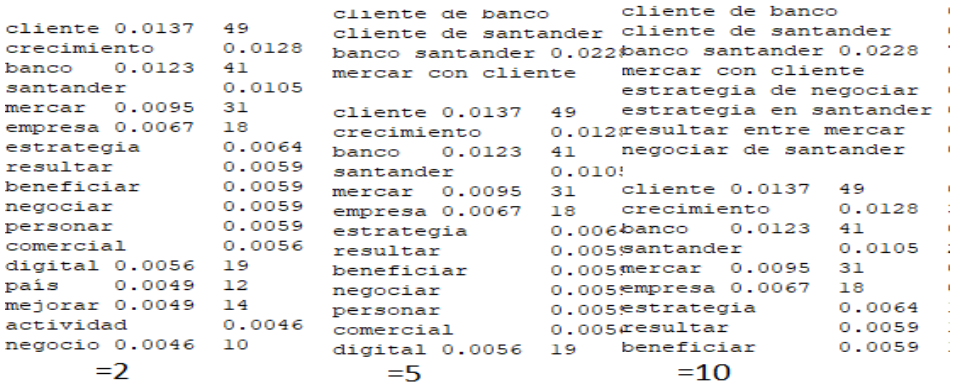


Figura 7: Resultados con variación del límite de nodos a comprobar (para multiwords)

Finalmente se presentando resultados de las comparativas de los dos analizadores sintácticos estudiados:

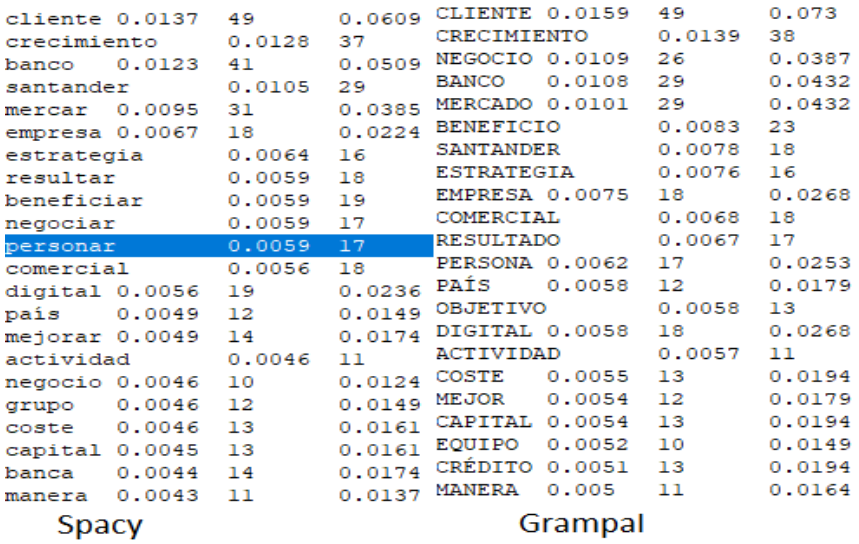


Figura 8: Resultados de los analizadores sintácticos (Spacy y Grampal)

Los dos tienen resultados consistentes, lo único destacable es que Spacy lematiza de una forma diferente. Por ejemplo, la raíz del término “mercado” para Spacy es “mercar” de ahí que aparezcan lemas que son verbos a pesar de haber realizado el filtrado sintáctico.

5.3.2 Etiquetado semántico

Retomando el ejemplo que se ha usado para la presentación de los resultados de la extracción de palabras clave (1-BancoSantander_2017) mostramos los resultados de las dos funciones disponibles relacionadas con el etiquetado semántico.

Se han estudiado los resultados producidos por diferentes valores de parametrización. Esto incluye variando el modo de acceso a los datos (atributo *mode*) y el modelo del incremento (atributo *model)*. Las variaciones del alcance (atributo reach) solo sirven para restringir el conjunto de datos.

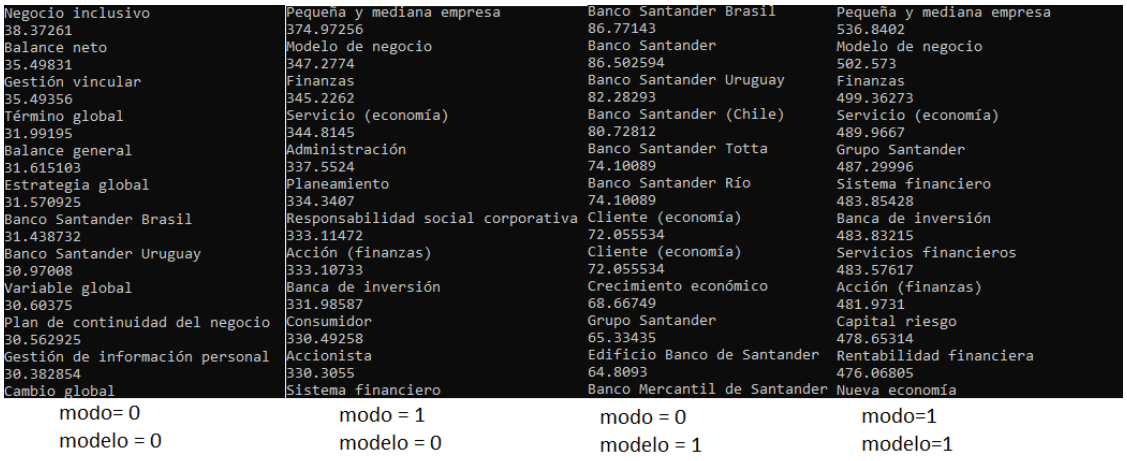


Figura 9: Resultados con variación de los atributos modo y modelo en búsquedas

La valoración de estos resultados es un tanto subjetiva, pero por lo general a lo largo del cuerpo de los más de 200 textos se han observado mejores resultados con la siguiente configuración:

modo de acceso 1, que accede a las apariciones del lema como concepto de otros lemas.

modelo de acceso 1, que implica el incremento exponencial explicado en la Sección 4.3.

Por último, mostramos los resultados variando el atributo de alcance, que limita el número de entradas tenidas en cuenta a la hora de realizar la búsqueda.

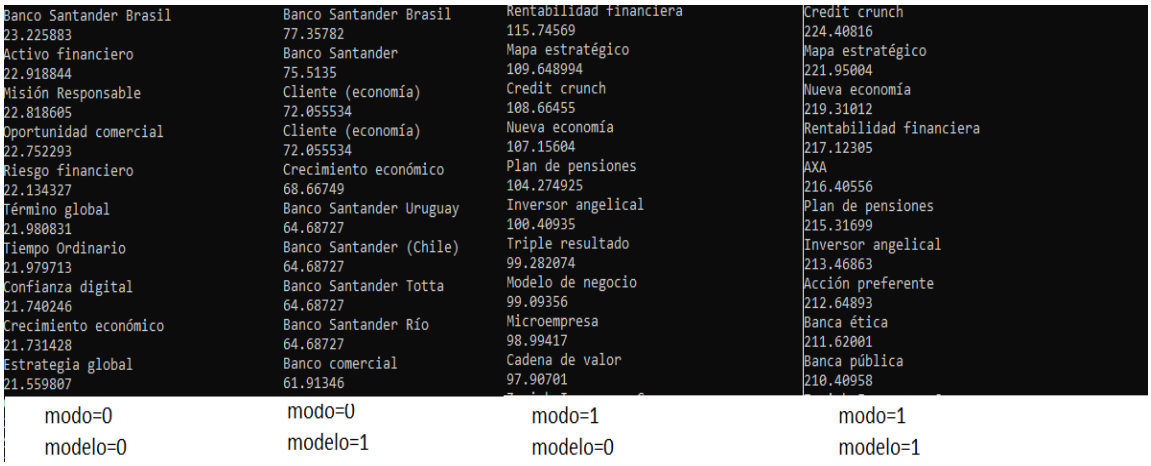


Figura 10: Resultados con variación de los atributos modo y modelo en búsquedas

Se puede observar que limitando la entrada de datos la aproximación mediante consulta directa del lema (modo ‘0’) da mejores resultados. De igual manera una aproximación lineal del incremento (modelo ‘1’) da mejores resultados. Esto se produce dado que su precisión depende en mayor medida (con respecto a las otras configuraciones) de la cantidad de datos a buscar.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

La creación de un sistema de recolección, etiquetado y recuperación de textos financieros ha sido una tarea complicado debido a la necesidad de integración de conocimientos en multitud de áreas. De hecho, sin la ayuda colaborativa de la comunidad especializada en el área del procesamiento de lenguaje y sus herramientas facilitadas este proyecto no habría sido posible.

Con este trabajo hemos conseguido una herramienta eficiente, capaz de trabajar con varias de las herramientas punteras en el ámbito del procesamiento de lenguaje. Adicionalmente, el sistema puede ser fácilmente ampliable debido a su diseño modular. Finalmente, las configuraciones ofrecidas hacen posible un refinamiento de los resultados.

Por todo ello, podemos concluir que los resultados obtenidos han sido satisfactorios, se ha creado un sistema que puede ser usado para el etiquetado semántico de la lengua español. Pero a pesar de los avances en la última década, la principal barrera de este proyecto ha sido la recuperación del repositorio semántico que sirve como eje central del proyecto. Debido a esta limitación los resultados no han podido alcanzar la precisión de algunos de los sistemas que trabajan con la lengua inglesa.

Como punto final destacar a título personal lo gratificante que ha sido desarrollar este proyecto multidisciplinar que me ha aportado tantos conocimientos como quebraderos de cabeza.

6.2 Trabajo futuro

Como trabajo futuro se puede continuar este proyecto con la implementación de nuevos algoritmos de extracción de palabras clave. Añadiendo entre otros algoritmos de agrupación y de incrustación se podría mejorar la precisión de los términos extraídos.

Con respecto a la segunda parte del proyecto, se podría mejorar el repositorio semántico extraído, corrigiendo entradas, mejorando las relaciones establecidas, añadiendo entradas…

Finalmente, en la parte de Elasticsearch se podrían desarrollar multitud de procesos para explotar esta herramienta tan poderosa.

Referencias

[1] “TFG Modelado de gramáticas basadas en dependencias a partir de un treebank de constituyentes, Rebeca de la Paz. Disponible: https://repositorio.uam.es/bitstream/handle/10486/680052/delaPaz_Gonzalez_Rebeca_tfg.pdf”

[2] “CoNLL-U Format, Disponible: https://universaldependencies.org/format.html”

[3] “Accuray, Precision, Recall or F1?, Koo Ping Shun,2018 Disponible: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9”

[4] “TextRank: Bringing Order into Texts, Rada Mihalcea and Paul Tarau, 200 Disponible: https://web.eecs.umich.edu/~mihalcea/papers/mihalcea.emnlp04.pdf”

[5] “Simple Unsupervised Keyphrase Extraction using Sentence Embeddings Kamil Bennani-Smires , Claudiu Musat , Andreaa Hossmann , Michael Baeriswyl , Martin Jaggi, 2018 Disponible: https://arxiv.org/pdf/1801.04470.pdf”

[6] “Key2Vec: Automatic Ranked Keyphrase Extraction from Scientific Articles using Phrase Embeddings, Debanjan Mahata, John Kuriakose Disponible: https://aclweb.org/anthology/N18-2100”

[7] “Unsupervised Keyphrase Extraction with Multipartite Graphs , Florian Boudin , Universidad de Nantes Disponible:https://www.aclweb.org/anthology/N18-2105”

[8] “Laboratorio de Lingüística Informática, 2016. Lllf.uam.es [En línea]. Disponible: http://www.lllf.uam.es/ESP/ [Último acceso: 2019]”

[9] “BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network, Roberto Navigli, Simone Paolo Ponzetto Disponible: http://wwwusers.di.uniroma1.it/~navigli/pubs/AIJ_2012_Navigli_Ponzetto.pdf#page=11”

[10] “TFG Adaptación del Stanford Parser al español, Borja Colmenarejo García Disponible https://repositorio.uam.es/bitstream/handle/10486/676885/Colmenarejo_Garcia_Borja_tfg.pdf”

[11] “Procesamiento semántico automático, enfocado en la coherencia textual, para apoyar la producción escrita de noticias, Sergio Hernándes Osuna, Anita Ferreira Cabreia” Disponible:https://scielo.conicyt.cl/scielo.php?script=sci_arttext&pid=S0071-17132016000200005”

[12] “Word Sense Disambiguation: The State of the Art, Nancy Ide, Jean Véronis Disponible: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.5205&rep=rep1&type=pdf”

[13] “Word Sense Disambiguation and induction, Leon Desczynski Disponible: https://es.slideshare.net/leonderczynski/word-sense-disambiguation-and-induction”

[14] “A State of the Art Technique in Semantic Analysis of Natural Language Utterances, Marija Brkić, Maja Matetić Disponible: https://pdfs.semanticscholar.org/c1f4/484423f223ba183af28b0efb0f8b70d79f2d.pdf”

[15] “Natural Language Processing of Semitic Languages, Dr. Imed Zitouni, Chap 4, p.140-145”

Glosario

Corpus: Conjunto amplio de ejemplos reales de uso de la lengua.

Iot: Internet of things, concepto que se refiere a la interconexión de objetos con internet.

Data Mining: Conjunto de procesos de extracción de información significativa de grandes conjuntos de datos.

Big Data: Conjunto de datos cuyo tamaño y complejidad sobrepasa las capacidades de procesamiento de los sistemas convencionales

Apache Lucene: API de código abierto centrada en la recuperación de información.

Machine Learning: Método de análisis de datos que automatiza la construcción de modelos análiticos.

TF-IDF: Medida numérica que expresa la relevancia de una palabra para un documento en una colección dada.

pip : Sistema de gestión de paquetes

JSON: Javascript Object Notation, formato de texto sencillo para el intercambio de datos.

Anexos

A Manual de instalación

La documentación relativa a la instalación del sistema de recolección, etiquetado y recuperación de textos financieros está disponible mediante el enlace a través del repositorio "https://github.com/Hugoferreiramoreno/TFG", pulsar en "Clone or Download" y a continuación "Download ZIP".
Una vez descargado el archivo, descomprimir, y obtener el archivo README.md, en el se indican los pasos a seguir para poder ejecutar nuestro proyecto.

B Manual del programador

La documentación del proyecto está disponible mediante el enlace a través del repositorio "https://github.com/Hugoferreiramoreno/TFG", pulsar en "Clone or Download" y a continuación "Download ZIP".
Una vez descargado el archivo, descomprimir, obtener la carpeta docs. Una vez dentro ir a build y luego elegiremos el formato de representación de la información (pdf o html). Elegiremos la carpeta correspondiente y dentro estára la información buscada.
Adicionalmente, se ha depositado un makefile junto a un .bat que al ejecutarlo genera la documentación del proyecto.